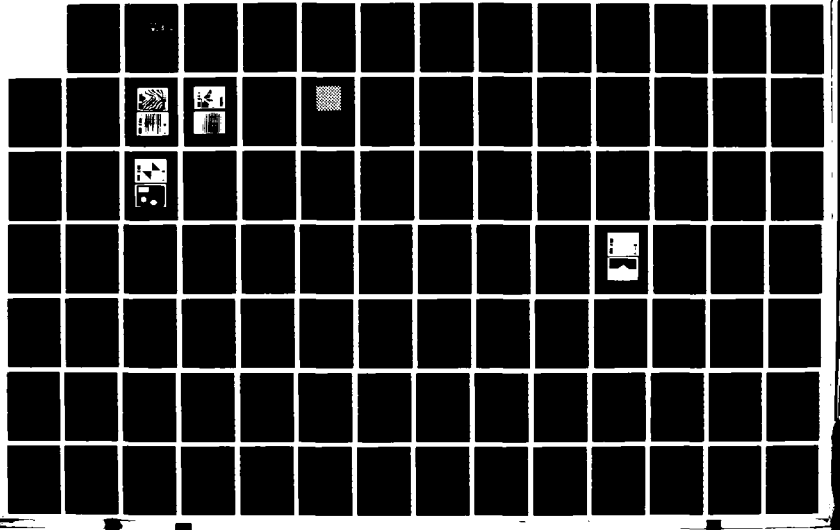
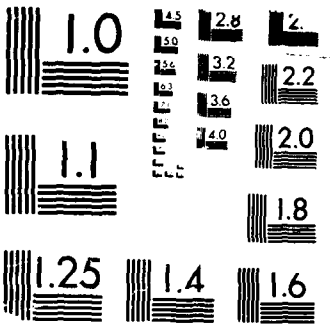
AD-A194 353 INVESTIGATION INTO THE USE OF TEXTURING FOR REAL-TIME COMPUTER ANIMATION(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA T W MEIER ET AL. MAR 88 NPS-52-88-003 1/2

UNCLASSIFIED F/G 12/6 NL





NPS52-88-003

NAVAL POSTGRADUATE SCHOOL

Monterey, California

INVESTIGATION INTO THE USE OF
TEXTURING FOR REAL-TIME COMPUTER ANIMATION

Timothy W. Meier

Robert B. McGhee

Michael J. Zyda

March 1988

Prepared for:
US Army Combat Developments Experimentation Center
Fort Ord, CA 93941

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NPS52-88-003	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (If applicable) 52	7a NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943	7b ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING / SPONSORING ORGANIZATION US Army Combat Developments Experimentation Ctr 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MIPR ATEC 88-86
8c. ADDRESS (City, State, and ZIP Code) Fort Ord, CA 93941	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
INVESTIGATION INTO THE USE OF TEXTURING FOR REAL-TIME COMPUTER ANIMATION

12 PERSONAL AUTHOR(S)
Meier, Timothy, W., McGhee, Robert B., Zyda, Michael J.

13a. TYPE OF REPORT	13b TIME COVERED FROM ____ TO ____	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
		1988, March	117

16. SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

We present in this study an investigation into the use of texturing on the Silicon Graphics, Inc. IRIS for real-time computer animation. Using a tool designed specifically for the IRIS for defining texture patterns, two approaches to the design and implementation of software functions to fill objects with multi-color texture patterns are discussed. The first approach makes use of the IRIS patterning hardware to fill objects with multi-color texture patterns. Realizing the limitations of the first approach, the second approach uses an algorithm to partition a polygon defined in three space into a number of smaller polygons, with each polygon representing a texture point.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED	
22a NAME OF RESPONSIBLE INDIVIDUAL Michael J. Zyda	22b TELEPHONE (Include Area Code) (408)646-2305	22c OFFICE SYMBOL 52Zk

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

☆ U.S. Government Printing Office: 1986—606-243

UNCLASSIFIED

Investigation into the Use of Texturing for Real-Time Computer Animation

Timothy W. Meier, Robert B. McGhee and Michael J. Zyda *

Naval Postgraduate School,

Code 52, Dept. of Computer Science,

Monterey, California 93943

ABSTRACT

We present in this study an investigation into the use of texturing on the Silicon Graphics, Inc. IRIS for real-time computer animation. Using a tool designed specifically for the IRIS for defining texture patterns, two approaches to the design and implementation of software functions to fill objects with multi-color texture patterns are discussed. The first approach makes use of the IRIS patterning hardware to fill objects with multi-color texture patterns. Realizing the limitations of the first approach, the second approach uses an algorithm to partition a polygon defined in three space into a number of smaller polygons, with each polygon representing a texture point.

‡ This work was supported by The US Army Combat Developments Experimentation Center, Fort Ord, California, the Naval Ocean Systems Center, San Diego and the Naval Postgraduate School's Direct Funding Program. This work was generated from Timothy W. Meier's Masters Thesis.

* Contact author.

TABLE OF CONTENTS

I. INTRODUCTION

A. TEXTURE DEFINITION

Texture is the surface detail or outward appearance of an object. In computer graphics, texturing is adding that surface detail to the component parts of a scene for the purpose of achieving greater realism. Texturing can be divided into two areas: color detail and surface roughness. Each area has been under considerable investigation [1-8]. For this study, we further divide texturing through color detail into two areas: random texture and texture patterns, with the more common being random texture. The color detail we see in a painting can be referred to as random texture as opposed to a texture pattern, since there is no smaller division of the object which characterizes the whole. In other words, there is no small pattern of colors that when repeated generates the whole picture. An example of a texture pattern might be the pattern created by a brick wall. A small pattern consisting of a section of brick when repeated both horizontally and vertically creates the whole wall. But if we look at the smallest detail in a brick, there are no two bricks that look alike and therefore no texture pattern exists. There are probably no true texture patterns that exist in nature. However, this does not prevent their use in computer graphics. Both random textures and texture patterns contribute towards the production of realistic computer generated images.

Texturing through color detail only results in a smooth surface still lacking in realism. The example commonly used is that of the imitation wood grain used on many

desks. The wood grain appears painted on the surface. The additional realism needed is achieved by adding surface roughness [6,7].

B. BACKGROUND

Adding color detail to a smooth surface is basically a mapping function. Mapping from a two dimensional texture space to three dimensional object space is a simple process, but difficulties arise in the transformation from object space to image space. With a polygon occupying a small area on a screen, many texture points are mapped to the same pixel on the screen. Textures are also very susceptible to aliasing.

The fact that adding texture to a smooth surface is a mapping function was first suggested by Catmull in 1974 [1]. Assuming a rasterized device, Catmull's algorithm involves subdividing a surface patch until a subpatch covers a single pixel center. At the same time, the texture space is subdivided similarily to arrive at a subpatch that defines an average intensity. This average intensity is a weighted average of the intensities within the texture subpatch. Blinn and Newell improved on Catmull's algorithm by using a different weighting function for calculating texture point intensities [2]. Crow introduced the use of pre-computed summed-area tables which reduced the computational expense of filtering [3].

Other approaches have also been tried. Schweitzer introduced a method of artificial texturing, making use of visual cues to approximate texture changes [4]. The goal was not to create realistic textured surfaces but to make use of texture gradients in surface visualization. Rather than project a texture on an object defined in three space, Samek, Slean and Weghorst introduced an algorithm to unfold a three-dimensional object in order to project the texture on a two-dimensional surface [5].

To add surface roughness to a textured object, Blinn introduced a method of perturbing the surface normal before intensity calculations [6]. The use of fractal surfaces to add surface roughness was originally applied by Carpenter, Fournier and Fussel. This technique subdivides a polygon, and then perturbs the locations of the midpoint of the polygon and the midpoints of its sides to produce a rough polygon surface [7].

Research in the area of real time texturing has also been conducted. Using a specially designed multi-processor system, Oka, Tsutsui, Ohba, Kurauchi and Tago have introduced real-time manipulation of texture mapped surfaces [8]. Using multi-processors, systems will no doubt be developed that can texture objects at a speed approaching real-time.

C. MOTIVATION

The motivation behind our research was to examine the potential for real-time texturing on a currently available graphics workstation, the Silicon Graphics IRIS-3120. The IRIS-3120 has some capabilities for real-time pattern fill of polygons. Our original thought was that we wanted to be able to use that real-time texturing capability to fill three-dimensional polygons with patterns appropriate to the scene at hand. In particular, we wanted to texture polygons in terrain scenes with appropriate textures such as grass, crops and others.

D. OVERVIEW

We began our look at real-time texturing with a focus on color detail texturing. The area chosen to investigate in color detail was that of adding texture patterns through the polygon fill operation. This choice was dictated by the capabilities existing on the IRIS.

If texture patterns were to be used, a tool had to be developed to easily produce those texture patterns. *Texted,* a color texture pattern editor, was developed to produce the desired texture patterns. The design of the editor was based on the fact that the IRIS had capabilities for pattern filling of polygons using patterns that are 16x16, 32x32, and 64x64 pixels in size. It was envisioned that the texture pattern editor could give the user the capability to design his own texture patterns or take texture patterns from digitized photographs. These texture patterns could then be used to fill polygons, and hence lead to more realistic images.

Once we had a texture editor, we had to develop functions to fill polygons with the defined texture patterns. Functions were developed to fill circles, arcs, rectangles and polygons with the multi-color texture patterns using the IRIS patterning hardware. Realizing the limitations of this method, functions were designed to fill arbitrary polygons defined in three space by using a polygon to represent each texture point. In this way, the IRIS Geometry Engines would handle the transformation of object space coordinates to image space and then to screen coordinates. A part of this design had to include a means of positioning the texture pattern on the polygonal surface. The design was limited to the use of polygonal surfaces, since for the most part, only polygons are used to define objects in real-time, three-dimensional animation on the IRIS.

II. TEXTED - A COLOR TEXTURE PATTERN EDITOR

A. OVERVIEW

Texted is a tool for the IRIS that produces 16x16, 32x32, or 64x64 multi-color texture patterns and stores them in the form of files. The original goal of the design was to provide the user a canvas on which he could paint a texture pattern, and then be able to save that pattern. Additional capabilities were added as their need or desireability became apparent. The capabilities of *texted* can best be described by the menus presented upon running the editor (Fig. 2.1, 2.2). There are five menus: the Canvas, Palette, Brush Stroke and Option menus.

B. PALETTE

When *texted* is executed, the Palette menu presents a grid of 300 different colors from which the user can choose. A selection labeled "T" is available should the user desire no color or transparency. As with most choices in *texted*, the user chooses a color by positioning the cursor over the desired color and depressing the right mouse button. The RGB values of the selected color are then displayed by the red, green and blue bars below the provided colors. Just below the provided colors and just above the red, green and blue bars is the selected color box, which always displays the current color.

The user also has the capability of defining his own colors by adjusting the red, green and blue bars. This is done by positioning the cursor in any of the bars and depressing RIGHTMOUSE. As long as RIGHTMOUSE remains depressed, the red, green or blue bar in which the cursor is positioned can be moved. The color defined is

displayed in the selected color box. This color is however, not fixed until RIGHTMOUSE is depressed with the cursor in the selected color box. Any square painted on the canvas with a color that is not fixed, changes color as any of the red, green, or blue bars is repositioned. A defined color can also be fixed if the user chooses a color from the provided 300 colors. This is to say that any square painted with the defined color, remains the same after a provided color is selected.

An additional capability in *texted* is to match any color displayed on the canvas, whether it be from a previously saved texture pattern or taken from a picture. The "match" mode is entered by positioning the cursor over the MATCH toggle at the bottom of the Palette menu. As long as the match toggle is on, a color can be matched from the canvas by positioning the cursor over the color and depressing RIGHTMOUSE. The color is displayed in the selected color box and its RGB values are reflected in the position of the red, green and blue bars. The current color cannot be used until the cursor is positioned over the MATCH toggle and RIGHTMOUSE is depressed to leave the match mode.

C. CANVAS

The Canvas menu is displayed on the left side of the screen, giving the user the options of a 16x16, 32x32, or 64x64 texture pattern. Initially a 16x16 texture pattern is displayed. A canvas size (referring to a texture pattern size) can be selected by positioning the cursor over the appropriate choice and depressing RIGHTMOUSE. The corresponding canvas is displayed. Any pattern displayed is lost by any new canvas selection unless the pattern is saved through the Option menu.

D. BRUSH STROKE

The Brush Stroke menu options are: BLOCK, HORIZONTAL, VERTICAL, RIGHT DIAGONAL, and LEFT DIAGONAL brush strokes. Brushes paint when the right mouse is depressed. Using the BLOCK brush stroke, the block in which the cursor is positioned is painted with the current color. As long as RIGHTMOUSE remains depressed, blocks are painted as the cursor is moved, giving the user a drawing capability. Using the HORIZONTAL brush stroke, every block is painted that is in the same row as the block in which the cursor is positioned. The VERTICAL brush stroke behaves the same, except that every block is painted that is in the same column as the block in which the cursor is positioned. Both RIGHT DIAGONAL and LEFT DIAGONAL paint every block that is in the same diagonal as the block in which the cursor is positioned. With all brush strokes, as long as RIGHTMOUSE remains depressed, and the cursor remains within the canvas, the canvas is painted using the current brush stroke and the current color.

E. OPTION

The options available in this menu are: VIEW, SAVE, RECALL, PICTURE and EXIT. All choices again are made by positioning the cursor over the selection and depressing RIGHTMOUSE. All prompts for file names are displayed below the canvas.

The VIEW option enables the user to view the texture pattern painted on the canvas by clearing the entire screen with the defined texture pattern. Each block of the texture pattern becomes a pixel, and the pattern is repeated until the screen is filled. The user returns again to texted by depressing RIGHTMOUSE. The screen then returns to the state it was in before making this selection.

The PICTURE option enables the user to select a texture pattern from a dithered picture file. The user is prompted for the file name, and the picture is displayed. If a dithered picture file does not exist or cannot be opened, texted returns to its previous state. Once the picture is displayed, the cursor takes on the form of a box to enable the user to select a 16x16, 32x32, or 64x64 pixel pattern. The cursor box size is changed by depressing MIDDLEMOUSE. A selection is made when RIGHTMOUSE is depressed. The pixel pattern within the cursor box becomes the current texture pattern when the main menu of texted is again displayed (Fig. 2.3).

The SAVE option allows the user to save the current texture pattern in a file and the RECALL option allows the user to recall any previously saved texture pattern. Again the user is prompted for a filename. If the file does not exist in the case of RECALL, or for some reason cannot be opened, the editor returns to its previous state. *Texted* is exited by the EXIT option.

F. OPERATION

Texted is run by entering "texted" at the keyboard. The initial state of the editor is a 16x16 pattern canvas, current brush stroke of BLOCK, MATCH toggle off, and a current color of WHITE. The upper left corner of the screen has an area labeled PATTERN which displays the current texture pattern as a 16x16, 32x32, or 64x64 pixel pattern. It is initially all white.

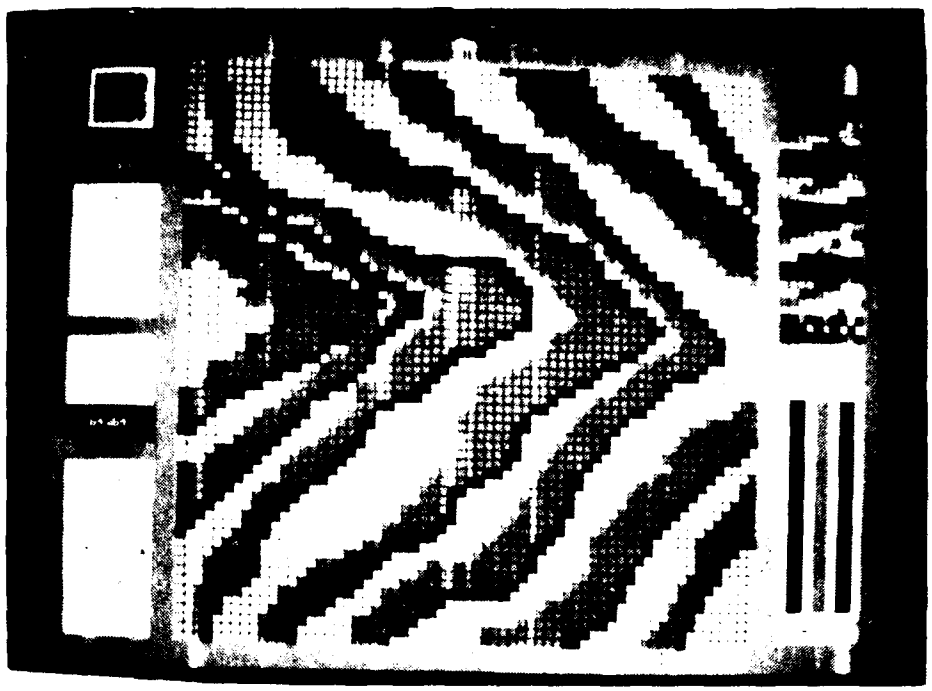


Figure 2.1 Main menu with wood texture pattern.

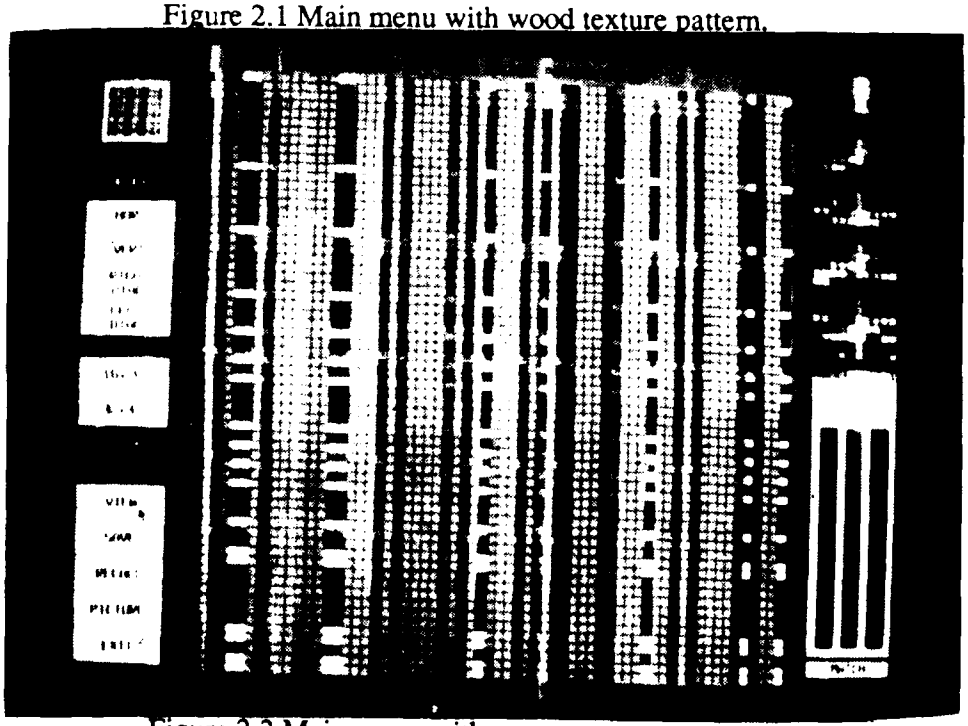


Figure 2.2 Main menu with weave texture pattern.

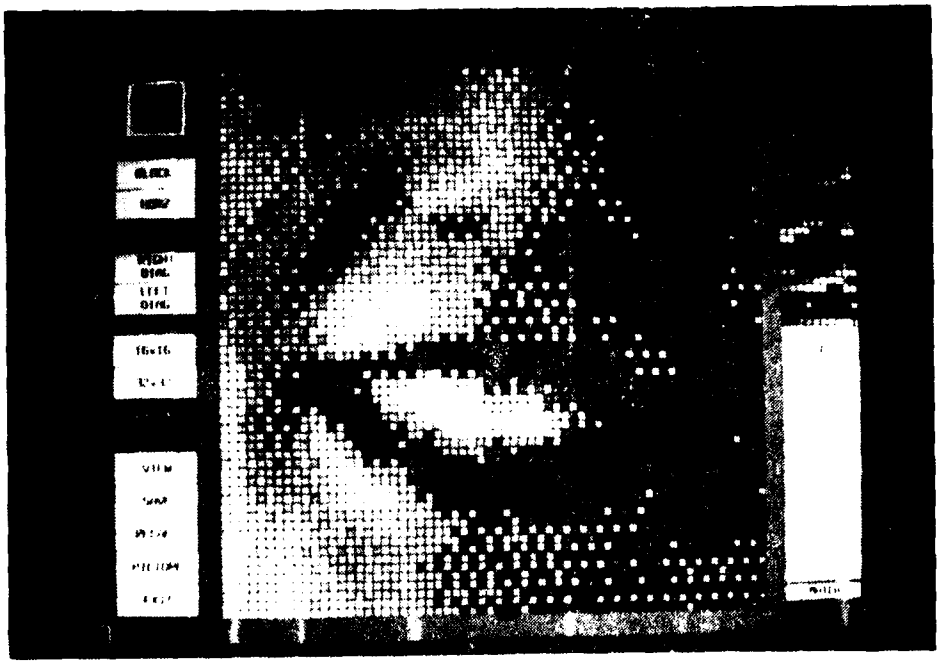


Figure 2.3 Texture pattern from dithered image.

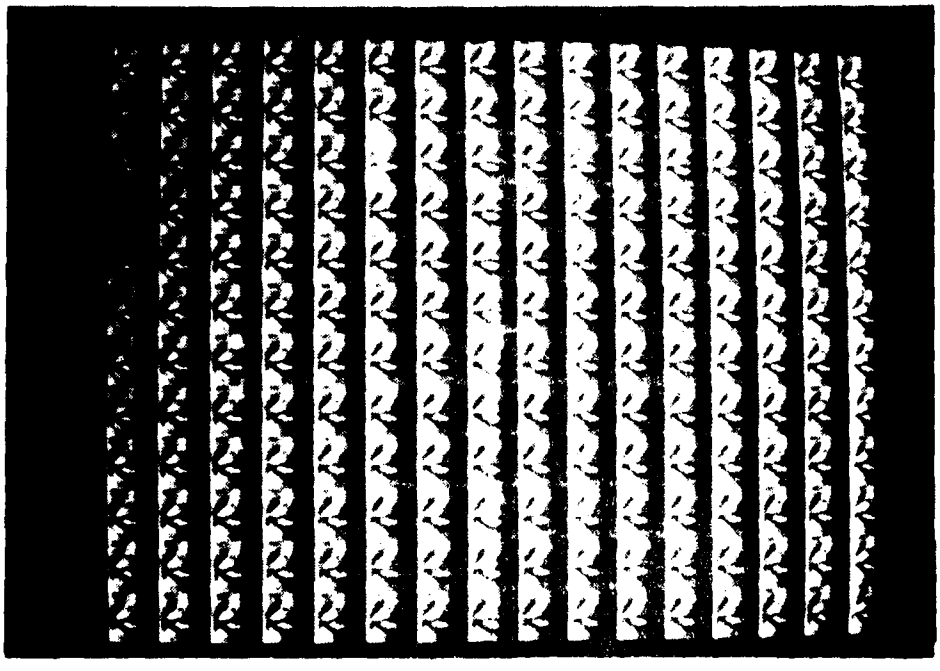


Figure 2.4 VIEW option with dithered image pattern.

III. TWO-DIMENSIONAL TEXTURING

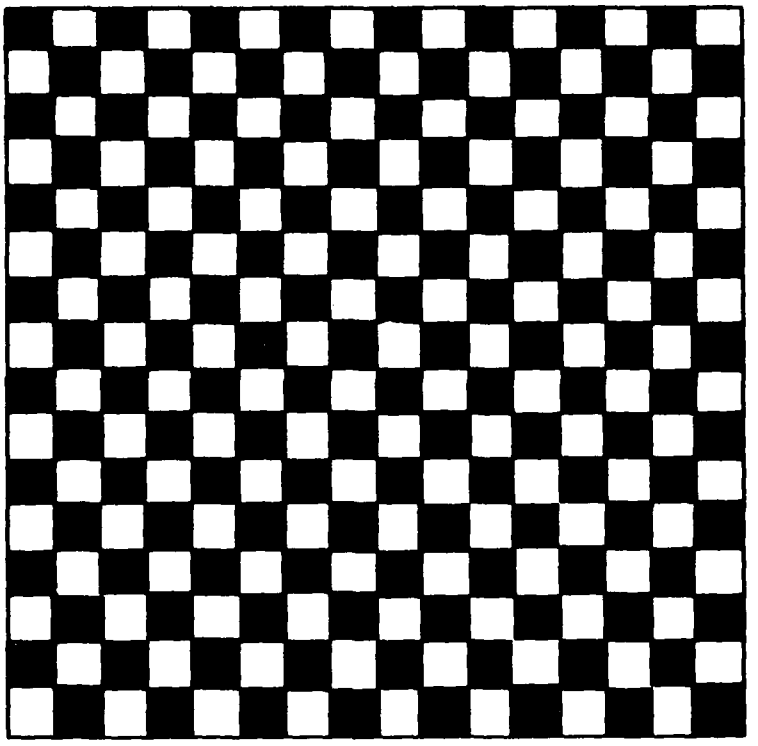
A. IRIS TEXTURE PATTERN FUNCTIONS

The IRIS has hardware capabilities for drawing a filled object with a defined texture pattern. A texture pattern is defined by a call to the system function

```
defpattern(n,size,mask)
short n,size;
short *mask;
```

which takes as input an index to a system table of patterns, a texture pattern size, and an array of short integers. A texture pattern can be a 16x16, 32x32 or 64x64 square pixel pattern and is specified by "size" taking on the value 16, 32 or 64. The texture pattern is an array of short integers that is a bit mask. The array of short integers controls which pixels are colored when a filled object is drawn. One row of a 16x16 texture pattern is specified by a short integer (Fig.3.1). One row of a 32x32 texture pattern is specified by two short integers, and one row of a 64x64 texture pattern is specified by four short integers. The bottom row starting with the left corner is specified first. The patterns are always aligned to the lower left corner of the screen so that two polygons filled with the same texture pattern and sharing a common edge appear continuous. The value of "n" specifies the index in the system table of patterns for the bit mask. The default pattern, which is a solid pattern, always has the index value of zero.

When a filled object is drawn with the IRIS, it is filled using the current texture pattern, color and writemask. A texture pattern defined with defpattern is made the



```
checkered=(0x5555,0xAAAA,0x5555,0xAAAA,
           0x5555,0xAAAA,0x5555,0xAAAA,
           0x5555,0xAAAA,0x5555,0xAAAA,
           0x5555,0xAAAA,0x5555,0xAAAA}
```

Figure 3.1 16x16 Texture Pattern.

current texture pattern with the function

setpattern(index)
short index;

which requires as input the index of the pattern in the system table of patterns. The system default texture pattern results in a solid filled object with the current color.

An example program using the IRIS texture pattern functions is shown in Figure 3.2. After some initialization calls, the pattern is defined with the system call

defpattern(1,16,checkered).

This results in the 16x16 bit mask "checkered" being saved with index 1 in the system table of patterns. The index 0 was not used since it is reserved for the default solid pattern. For this example the background color is made white with

color(WHITE);
clear().

The texture pattern "checkered" becomes the current texture pattern with the system call

```
main()
{
  .
  .
  .
 ginit();                        /* Initialize IRIS. */
  .
  .
 defpattern(1,16,checkered);     /* Define texture pattern. */
  .
  .
 color(WHITE);                   /* Clear screen to white. */
 clear();
  .
  .
 setpattern(1);                  /* Set texture pattern. */
 color(RED);
 rectf(10.0,10.0,20.0,20.0);     /* Rectangle drawing primitive. */
  .
  .
 gexit();
}
```

Figure 3.2 Example program for IRIS texture pattern functions.

```
setpattern(1);
```

and finally the rectangle is drawn by

```
color(RED);
rectf(10.0,10.0,20.0,20.0).
```

The bit mask "checkered" determines which pixels are colored red when the rectangle is filled. Those pixels not colored red, remain the background color of white.

B. TEXTURE PATTERNS AS OVERLAYS

With the standard IRIS texture support, one can write a program that fills objects with a single-color bit mask. We had a desire for filling objects with multiple color bit masks that could be defined by *texted*. Using the IRIS's hardware facilities for texture patterns, we developed a method for filling objects with multi-color texture patterns.

As described earlier, a texture pattern can be represented as an array of short integers that control which pixels are colored when a rectangle, circle, arc or polygon is filled. A multi-color texture pattern can easily be achieved by repeating the drawing primitive as many times as there are colors in the multi-color texture pattern, with a different system defined texture pattern used for each color.

As an example, suppose there are three colors desired in a 16x16 multi-color texture pattern. Three different system texture patterns or bit masks would have to be defined, one for each color in the multi-color texture pattern. Each bit in the multi-color texture pattern would be set only once by one of the bit masks. The drawing primitive has to be repeated three times, each time using a different system defined texture pattern and a different color.

C. USE OF TEXTED

The texture patterns defined by *texted* are saved as a file consisting of the pattern size, the number of colors used in the multi-color texture pattern, and the RGB values of the colors used along with an array of integers that describe the bit mask for that particular color (Fig. 3.3). The file name is used when defining the multi-color texture pattern.

D. MULTI-COLOR TEXTURE PATTERN FUNCTIONS

Functions were developed to parallel the IRIS texture pattern functions and drawing primitives that make use of the system defined texture patterns. A comparison of the IRIS functions and the new functions is shown in Table 3.1.

```
16
3
255 0   0
aaaa 5555 aaaa 5555 aaaa 5555 aaaa 5555
aaaa 5555 aaaa 5555 aaaa 5555 aaaa 5555

109 218 0
5555 2222 5555 8888 5555 2222 5555 8888
5555 2222 5555 8888 5555 2222 5555 8888

0  0  0
0   8888 0   2222 0   8888 0   2222
0   8888 0   2222 0   8888 0   2222
```

Figure 3.3 Texted output file.

To define a multi-color texture pattern, the function

```
defcolorpattern(first_pattern,first_color,filename)
short first_pattern;
Colorindex first_color;
char filename;
```

is used as opposed to the IRIS defpattern function. The user must provide as input to the function the first index in the system table of patterns to use, the beginning index in the color map to use, and finally the name of the file containing the multi-color texture pattern (Fig. 3.3). For every color used in the multi-color texture pattern, there exists a corresponding system table of patterns index as well as a color map index. The function *defcolorpattern* calls the IRIS function defpattern and makes a color map entry for each color used in the multi-color texture pattern. The implementation of *defcolorpattern* is shown in Figure 3.4. The number of colors used in the multi-color texture pattern is returned by the function *defcolorpattern*. This value can be used to determine the next available index in the system table of patterns and in the color map.

Table 3.1 Comparison of IRIS texture functions and new functions.

IRIS	NEW
defpattern	defcolorpattern
setpattern	-
rectf	rectcolorf
polf	polcolorf
arcf	arccolorf
circf	circcolorf
clear	colorclear

The functions that draw filled objects with the defined multi-color texture patterns require as input the beginning index in the system table of patterns, the beginning index in the color map, the number of colors used in the texture pattern and finally the drawing

parameters. The implementation of each function is demonstrated with the new function *rectcolorf* shown in Figure 3.5. The function repeats the loop as many times as there are colors in the texture pattern. With each loop, the drawing primitive is repeated with a new system defined texture pattern and a new color. Specifications for all multi-color texture pattern functions can be found in Appendix A.

E. EXAMPLE

In Figure 3.6, a program fragment is shown that demonstrates the use of the new multi-color texture pattern functions. The complete program can be found in Appendix B. The program fills a rectangle, circle, arc and polygon with multi-color texture patterns stored in the files "brick.pat", "wood.pat", "weave.pat" and "mod.pat". Each file was produced by the color texture pattern editor *texted*.

After some IRIS initialization and variable definitions, four different multi-color texture patterns are defined with the new function *defcolorpattern*. Since the IRIS's default solid texture pattern uses the index 0 in the system table of patterns, the first index in the system table of patterns to be used by the first texture pattern is 1. The first color map index to be used is 8 since the first eight indices (0-7) are system default colors. The numbers 1 and 8 are therefore the first two inputs followed by the file name "brick.pat" in the function call

n=defcolorpattern(1,8,"brick.pat").

The number of colors used in the multi-color texture pattern saved in "brick.pat", which also corresponds to the number of indices used in the system table of patterns to define that multi-color texture pattern, is returned by the function and assigned to the variable

```
defcolorpattern(first_pattern,first_color,filename)
short first_pattern;
Colorindex first_color;
char filename[];
{
 short number_of_colors, next_pattern, pattern_size, bitword;
 short red_value, green_value, blue_value, i,k;
 short mask[256];
 Colorindex next_color;
 FILE *fp,*fopen();
 fp=fopen(filename,"r");

 /* Read pattern size and number of colors used from file. */
 fscanf(fp,"%hd%hd",&pattern_size,&number_of_colors);

 next_pattern=first_pattern;
 next_color=first_color;

 /* Repeat for each color used in the multi-color texture pattern. */
 for (k=1; k<=number_of_colors; k++){

  /* Read the RGB values defining the color and place in color map. */
  fscanf(fp,"%hd%hd%hd",&red_value,&green_value,&blue_value);
  mapcolor(next_color,red_value,green_value,blue_value);

  next_color+=1;       /* Advance color map index. */

  /* Read bit mask. */
  for (i=0; i<(pattern_size*(pattern_size/16)); i++){
   fscanf(fp,"%hx",&bitword);
   mask[i]=bitword;
  }
  /* Define the bit mask in the system table of patterns. */
  defpattern(next_pattern,pattern_size,mask);

  next_pattern+=1;      /* Advance system table of patterns index. */
 }
 fclose(fp);
 return(number_of_colors);
}
```

Figure 3.4 Function to define multi-color texture pattern.

```
rectcolorf(first_pattern,first_color,number_of_colors,x1,y1,x2,y2)
short first_pattern;
Colorindex first_color;
short number_of_colors;
Coord x1,y1,x2,y2;
{
 short i;
 short next_pattern;
 Colorindex next_color;

 pushattributes();
 next_pattern=first_pattern;
 next_color=first_color;

 /* Repeat for each color used in multi-color texture pattern. */
 for (i=1; i<=number_of_colors; i++){
  setpattern(next_pattern);    /* Set current texture pattern. */
  next_pattern+=1;             /* Advance system table of patterns index. */
  color(next_color);           /* Set current color. */
  next_color+=1;               /* Advance color map index. */

  /* Draw rectangle using current color and current texture pattern. */
  rectf(x1,y1,x2,y2);
 }

 popattributes();
}
```

Figure 3.5 Function to fill a rectangle with multi-color texture pattern.

"n". The value assigned to "n" is then used to calculate the next available system table of patterns index and the next available color map index when the multi-color texture pattern saved in "wood.pat" is defined with

m=defcolorpattern(1+n,8+n,"wood.pat").

This defcolorpattern call returns to the variable "m" the number of colors used by the texture pattern saved in "wood.pat", which is then used when defining the multi-color

texture pattern saved in "weave.pat":

p=defcolorpattern(1+n+m,8+n+m,"weave.pat").

Finally, the value assigned to "p" is used to define the multi-color texture pattern saved in "mod.pat" by the function call

q=defcolorpattern(1+n+m+p,8+n+m+p,"mod.pat").

After the four multi-color texture patterns are defined, the screen is cleared to white using the system default solid texture pattern and then the four filled objects are drawn. Since the rectangle drawn uses the texture pattern saved in file "brick.pat", it must take as inputs, the first index in the system table of patterns which defines that pattern, the first index in the color map used by that pattern, and the number of colors in the texture pattern. The first two inputs correspond to the same inputs used when defining the texture pattern. The third input was returned by the defcolorpattern function taking the file "brick.pat" as an input. The function call to fill a rectangle with the texture pattern saved in "brick.pat" is

rectcolorfi(1,8,n,75,500,425,700).

The remaining arguments are the drawing parameters normally used with the system function call rectf. The function call

circcolorfi(1+n,8+n,m,250,250,100);

fills a circle with center at (250,250) and radius equal to 100, with the texture pattern saved in "wood.pat" since the first two arguments match the first two arguments in the defcolorpattern call that defined that multi-color texture pattern. The value of "m" was assigned by the same function call. An arc is filled with the multi-color texture pattern

```
main()
{
   .
   .
   .
  ginit();                    /* Initialize the IRIS. */
   .
   .
   .
  /* Define multi-color texture patterns. */
  n=defcolorpattern(1,8,"brick.pat");
  m=defcolorpattern(1+n,8+n,"wood.pat");
  p=defcolorpattern(1+n+m,8+n+m,"weave.pat");
  q=defcolorpattern(1+n+m+p,8+n+m+p,"mod.pat");
   .
   .
  color(WHITE);               /* Clear screen to white. */
  clear();
   .
   .
  rectcolorfi(1,8,n,75,500,425,700);          /* Draw a rectangle. */
  circcolorfi(1+n,8+n,m,250,250,100);         /* Draw a circle. */
  arccolorfi(1+n+m,8+n+m,p,550,500,200,300,900); /* Draw an arc. */
  polcolorf2i(1+n+m+p,8+n+m+p,q,6,a);         /* Draw a polygon. */
   .
   .
   .
  gexit();
}
```

Figure 3.6 Example program using multi-color texture pattern functions.

saved in "weave.pat" by

arccolorfi(1+n+m,8+n+m,p,550,500,200).

The arc has a center point at (550,500) and a radius of 200. The polygon is filled with the multi-color tetxure pattern saved in "mod.pat" by

polcolorf2i(1+n+m+p,8+n+m+p,q,6,a).

The array "a" defines the polygon and has 6 points. Figure 3.7 is the display generated by this example program.

F. LIMITATIONS

The use of this method for drawing objects with multi-color texture patterns is very limited. Though described as 2D texturing in this chapter, the IRIS hardware facilities for filling objects with texture patterns can also be used to fill 3D objects. Unfortunately, texture patterns are fixed to the screen mask. When an object is rotated or translated, the same pixel remains the same color as long as it remains within the object in screen coordinates. As our filled object is rotated or translated, this gives the appearance that the texture pattern does not translate or rotate with the object.

The algorithm for filling objects with multi-color texture patterns repeats the drawing primitive as many times as there are colors in the multi-color texture pattern. Thus the performance of the algorithm is at least as many times slower as there are colors in the texture pattern.

Another problem encountered is the amount of memory available to store the system defined texture patterns. An option was added to texted to take texture patterns from dithered images. After only defining a few multi-color texture patterns taken from dithered images, the available memory for the system table of patterns was exceeded.

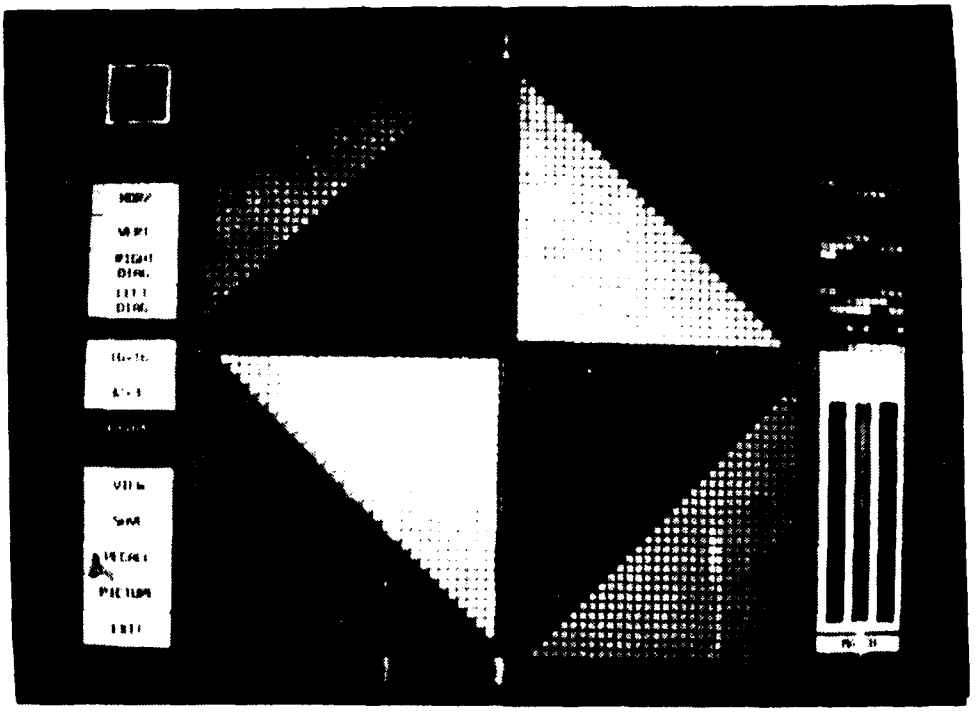


Figure 3.6 Pattern used in demo1.c

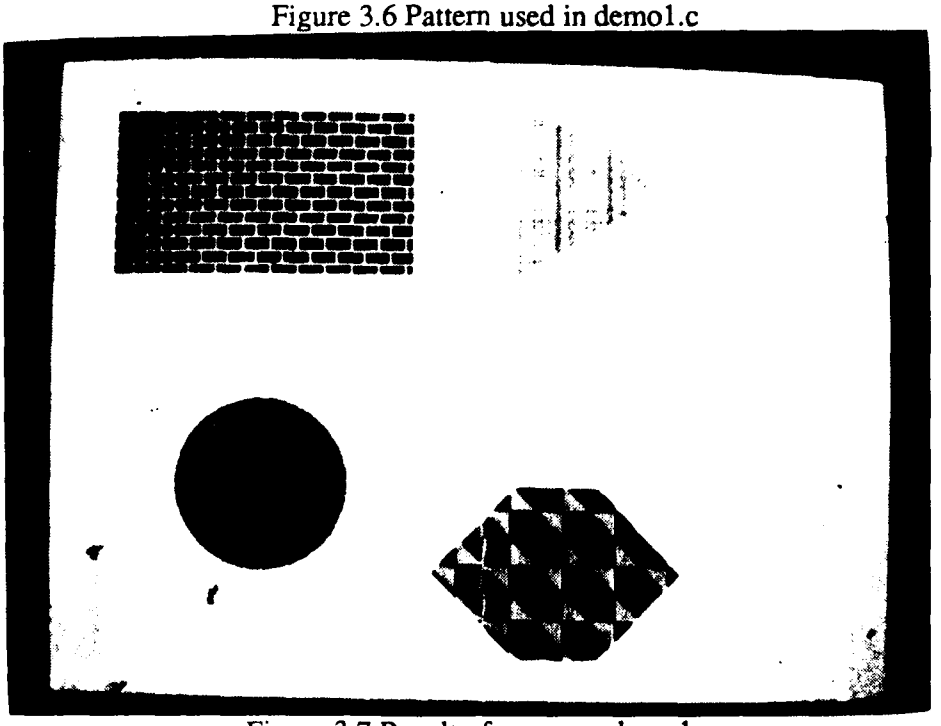


Figure 3.7 Result of program demo1.c.

IV. THREE-DIMENSIONAL TEXTURING

To fill 3D polygons with texture patterns that are not fixed to the screen mask, a system was implemented that partitions a polygon into a number of smaller polygons (texture point polygons), each of which represents a point in our texture pattern. With each texture point represented as a polygon, all the IRIS capabilities for the transformation and filling of polygons are utilized. This includes the IRIS's implementation of backface polygon removal for hidden surface elimination. Whereas the IRIS texture pattern facilities define a 16x16, 32x32 or 64x64 bit mask, the system implemented fills a polygon with a 16x16, 32x32 or 64x64 texture pattern where the size of each texture point polygon is set by the user. What was a pixel in a texture pattern before, now becomes a polygon.

A. GENERAL DESCRIPTION

The system was designed so that the same files produced by *texted* could also be used by our 3D polygon fill functions. The system was also designed so that a texture pattern could be placed on a polygon defined in three space, in any orientation desired by the user. This gives the user a sort of wallpapering capability with the polygon defined in three space. The user must insure that the texture pattern edges match where polygon edges meet. Finally, realizing that real-time texturing of polygons becomes unrealistic as texture point polygons become smaller, a feature was implemented to enable the user to define the size of each texture point polygon.

The implementation of this system can be explained by the major functions it must perform. To facilitate placement of the texture pattern on the polygon defined in three space, the polygon is first mapped to the x-y plane. A counterclockwise ordering of the polygon's coordinates specifies the face of the polygon on which the texture pattern is to be placed. The positioning of the texture pattern on the polygon is achieved partly by this mapping, and also by the reference point of the texture pattern set by the user. The texture point polygons are calculated by using an algorithm for clipping an area with a line. The texture point polygons are finally mapped individually back to their proper relative position in three space before being drawn.

B. MAPPING A POLYGON TO THE X-Y PLANE

The first step in the texturing of a polygon defined in three space is to map the object space coordinates of the polygon to the x-y plane. This mapping is performed by calculating transformation matrices for translating the first point of the polygon to the origin, and for rotating the polygon about the x, y, and z axes. At the same time, the inverse is calculated for each transformation matrix to be used in mapping each texture point polygon to its proper relative position in three space.

1. Normal to a Polygon

To calculate the rotations necessary for positioning an arbitrary polygon in the x-y plane, the equation of the plane and the normal to that plane are calculated. Three non-colinear points or vertices of the polygon to be mapped are selected to define the equation of the plane for the 3D polygon, and the normal to that plane.

$$A_x+B_y+C_z=D;$$

$$\vec{N}=A\vec{i}+B\vec{j}+C\vec{k};$$

The coefficients for the equation of the plane and the normal to the plane can be calculated using the following equations:

$$A=(y_0-y_1)(z_2-z_1)-(y_2-y_1)(z_0-z_1);$$

$$B=(z_0-z_1)(x_2-x_1)-(z_2-z_1)(x_0-x_1);$$

$$C=(x_0-x_1)(y_2-y_1)-(x_2-x_1)(y_0-y_1);$$

D can be determined using the coefficients A, B, and C along with one point of the polygon. The direction of the normal to the polygon is determined by the ordering of the vertices in the array that defines the polygon in three space.

2. Rotation Angles About the Axes

With the equation of the plane determined, all vertices of the polygon are checked to insure the polygon is planar. If the polygon is planar, the angles necessary to rotate the polygon into the x-y plane are calculated.

The counterclockwise ordering of the polygon points not only determines the direction of the normal to the plane, but it also determines on which face of the polygon the texture pattern is to be placed. The counterclockwise ordering of the polygon vertices must be maintained as it is positioned in the x-y plane.

To calculate the rotations about the axes necessary to bring the polygon into the x-y plane, the angles between the normal and the axes must be calculated. These angles are calculated using the following equations and are shown in Figure 4.1.

$$\cos\alpha=\frac{A}{\sqrt{A^2+B^2+C^2}}$$

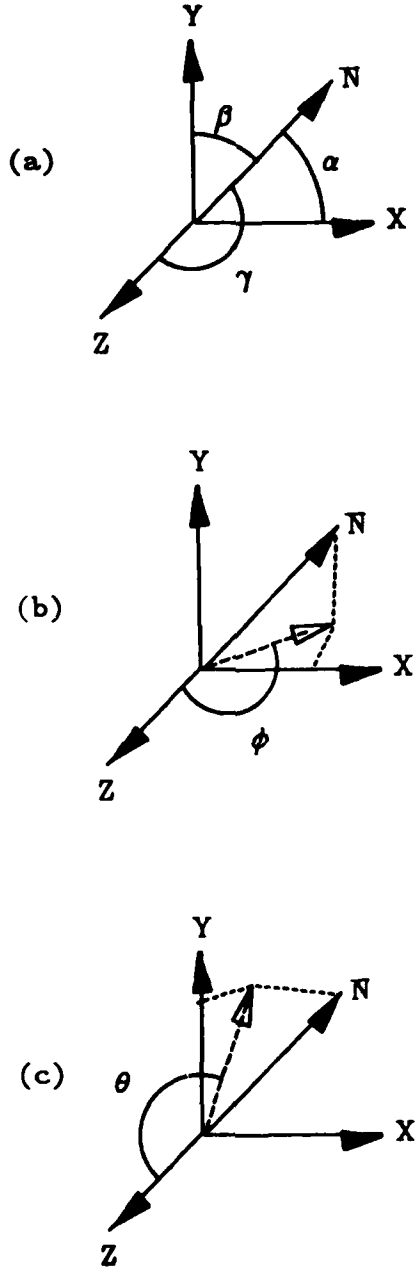


Figure 4.1 Rotation angles about x and y axes.

$$\cos\beta = \frac{B}{\sqrt{A^2+B^2+C^2}}$$

$$\cos\gamma = \frac{C}{\sqrt{A^2+B^2+C^2}}$$

For the polygon to lie in the x-y plane with the coordinates ordered in a counter clockwise fashion, the angle between the normal and the z axis must be 180 degrees.

To calculate the necessary angles of rotation, first the dot product of the projected $\vec{N}$ and the z axis is divided by the product of the magnitude of the two vectors, yielding the cosine of the angle ϕ about the y axis.

$$\cos\phi = \vec{N}' \frac{\vec{Z}}{|\vec{N}'||\vec{Z}|}$$

If the angle α is less than 90 degrees, the angle of rotation necessary about the y axis is $180-\phi$. If the angle α is greater than 90 degrees, the angle of rotation necessary about the y axis is $\phi-180$.

A second projection of the normal onto the y-z plane is made and the angle θ about the x axis is calculated in a similar manner. This time, if the angle β is less than 90 degrees, the angle of rotation necessary about the x axis is $\theta-180$. If the angle β is greater than 90 degrees, the angle of rotation necessary about the x axis is $180-\theta$.

3. Transformation

Once the angles of rotation necessary about the x and y axes are calculated, transformation matrices are used to map the vertices of the polygon to the x-y plane. The inverse transformation matrix in each case is calculated for mapping the texture point polygons to their proper relative position in three space. We translate the polygon to the origin by adding the negative x, y, and z value of the first vertex. Letting l,

m, and n represent the x, y, and z values respectively, the following transformation matrix along with its inverse is developed:

$$T_1=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -l & -m & -n & 1 \end{bmatrix}$$

$$T_1^{-1}=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ l & m & n & 1 \end{bmatrix}$$

Using the rotation angle ϕ about the y axis, the following transformation matrix along with its inverse is developed:

$$T_2=\begin{bmatrix} \cos\phi & 0 & -\sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_2^{-1}=\begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Using the rotation angle θ about the x axis, the following transformation matrix along with its inverse is developed:

$$T_3=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_3^{-1}=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To aid in positioning the texture pattern on the face of the polygon, a third angle of rotation about the z axis is used, with zero degrees positioning the first two points of the polygon on the x axis. This angle of rotation is explained further in the next section on texture pattern positioning. It is set by the user. Using the rotation angle ψ about the z axis, the following transformation matrix along with its inverse is developed:

$$T_4=\begin{bmatrix} \cos\psi & \sin\psi & 0 & 0 \\ -\sin\psi & \cos\psi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_4^{-1}=\begin{bmatrix} \cos\psi & -\sin\psi & 0 & 0 \\ \sin\psi & \cos\psi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The overall transformation matrix for mapping the polygon defined in three space to the x-y plane is achieved with the following expression:

$$T=T_1T_2T_3T_4$$

The new polygon P' in the x-y plane, is achieved by applying the overall transformation T to the old polygon P defined in three space.

$$P'=PT$$

Since each texture point polygon must be mapped back to its proper relative position in object space, an overall inverse transformation matrix is calculated using the inverse matrices of each translation and rotation.

$$T^{-1}=T_4^{-1}T_3^{-1}T_2^{-1}T_1^{-1}$$

A reverse mapping can be achieved as follows:

$$P=P'T^{-1}.$$

C. TEXTURE PATTERN POSITIONING

Positioning of a multi-color texture pattern on a polygon defined in three space is achieved by a combination of actions. It begins with the mapping of the original polygon to the x-y plane. This mapping to the x-y plane puts the polygon defined in three space in a reference which is easily visualized by the user and makes the calculation of texture point polygon coordinates simpler.

A texture pattern displayed as a matrix of colors in *texted* can be thought of as a grid of clipping lines which is repeated to partition a polygon into texture point polygons. The lower left corner of the texture pattern is called the reference point and is the point from which the clipping lines are repeated. Four clipping lines along with a color define a filled texture point polygon. The texture point polygon size specified by the user determines the spacing of the clipping lines. Positioning a texture pattern on a polygon involves superimposing this grid of clipping lines over the polygon in the x-y plane. It

can easily be seen how the specification of the reference point and the orientation of the polygon in the x-y plane affect the partitioning of that polygon into texture point polygons (Fig. 4.2).

A counterclockwise ordering of the polygon coordinates in three space specifies the face of the polygon on which the texture pattern is to be placed. This is the first means by which the user controls the texture pattern positioning.

The polygon defined in three space is always mapped to the x-y plane with the first point at the origin and the position of the second point determined by the rotation angle ψ about the z axis specified by the user. A rotation angle of zero degrees places the second point of the polygon on the x axis. The angle ψ is measured from the x axis and positive angles describe counterclockwise rotations. The user can control the orientation of the polygon in the x-y plane by the angle ψ and the specification of the first point of the polygon. This is the second means by which the user controls texture pattern positioning (Fig.4.3).

The final means by which the user controls positioning of the texture pattern is the specification of the pattern reference point. After the original polygon is positioned in the x-y plane, the system developed determines the upper and lower limits of the texture pattern. The lower limit of the texture pattern is the minimum x and y value of all polygon coordinates, and the upper limit is the maximum x and y values of all polygon coordinates. The reference point of the texture pattern, if not changed by the user, is the lower limit. The user has the ability to shift the reference point, which has the affect of shifting the texture pattern right or left, up or down. This is important when trying to

make pattern edges match where polygon edges meet. Positioning a texture pattern on a polygon is demonstrated in Figure 4.4.

D. CALCULATION OF TEXTURE POINT POLYGONS

The calculation of texture point polygon coordinates is done by using an algorithm for clipping an area with a line [9]. This algorithm takes the coordinates of a polygon and the equation of a line and finds the points of intersection. All points to one side of the line are visible, while all others are clipped. The points of intersection along with the polygon coordinates that are visible become the coordinates of a new polygon.

Starting with the lower limit of the texture pattern and the increment size specified by the user, two vertical clipping boundaries and two horizontal clipping boundaries, both an "increment" apart are calculated.

vertical clipping boundaries:

$$x = x_{lowerlimit};$$

$$x = x_{lowerlimit} + increment;$$

horizontal clipping boundaries:

$$y = y_{lowerlimit};$$

$$y = y_{lowerlimit} + increment;$$

The clipping boundaries, a color index from the array that defines the texture pattern and the overall polygon coordinates determine the coordinates and the color of the first texture point polygon. The resulting polygon after application of the clipping boundaries, is a polygon that is visible within the box formed by the four clipping boundaries. If this box is completely within the overall polygon, the texture point polygon becomes a square

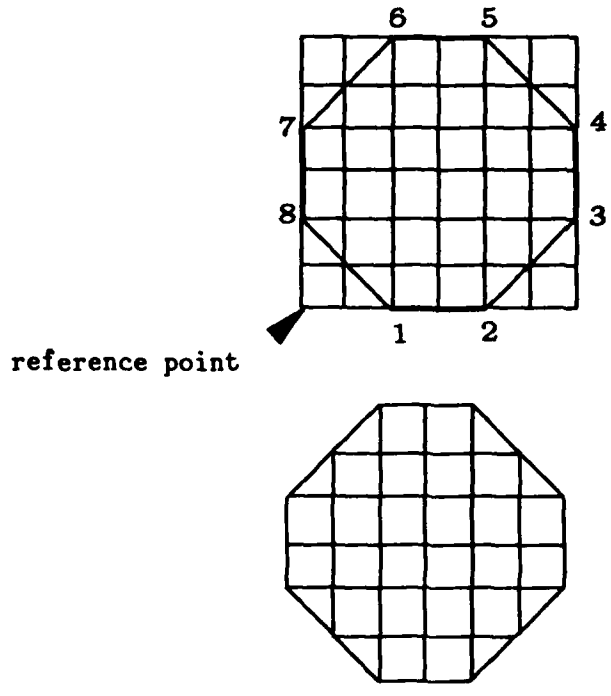


Figure 4.2 Partitioning of a polygon by a texture pattern.

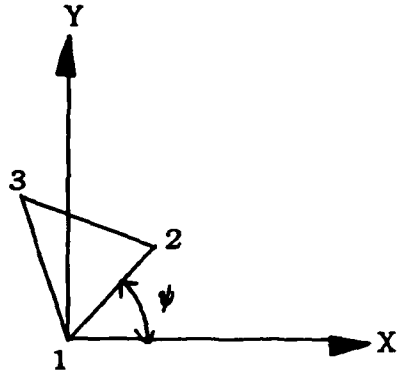
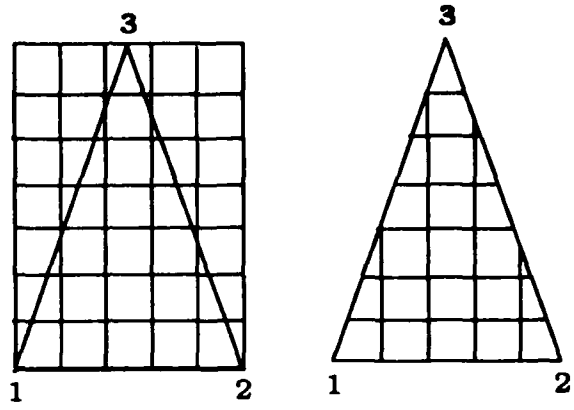
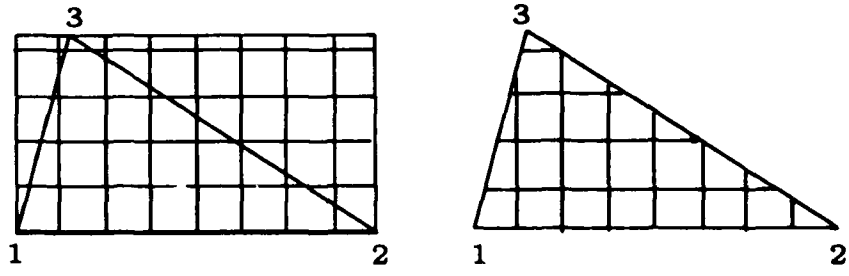


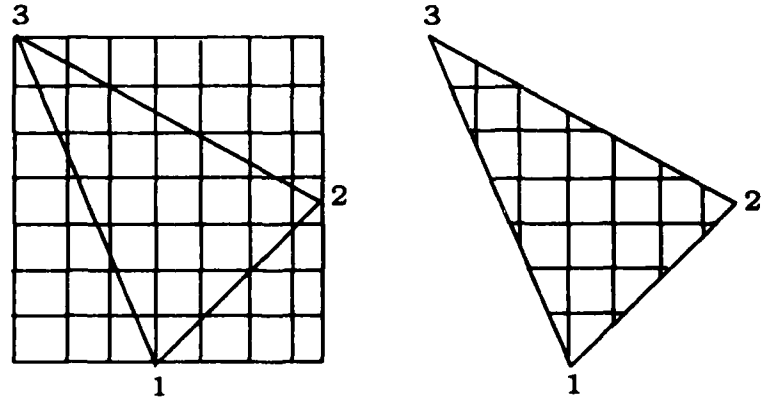
Figure 4.3 Rotation angle about z axis.



(a) Initial pattern positioning.



(b) Change ordering of coordinates.



(c) Rotation about s axis.

Figure 4.4 Texture pattern positioning.

polygon. If the box formed is only partially within the overall polygon, the texture point polygon becomes a polygon defined by the edges within the box and the box corners that are within the overall polygon (Fig. 4.5).

The array of color indices that define the texture pattern provide the color index when filling a texture point polygon. By changing the beginning indices to reference the texture pattern array, the appearance of shifting the reference point is achieved. The coordinates for the texture point polygons are calculated in the x-y plane. The overall inverse transformation matrix calculated when positioning the polygon in the x-y plane is used to map the coordinates of the texture point polygon to its proper relative position in object space before it is filled.

E. MULTI-COLOR TEXTURE PATTERN FUNCTIONS

Functions were developed to parallel the IRIS texture pattern function defpattern and the drawing primitive polf for filling a polygon in three space. To define a multi-color texture pattern to be used in three space, the function

```
defcolorpattern3(first_color,pattern,filename)
Colorindex first_color;
Colorindex pattern[64][64];
char filename[];
```

is used as opposed to the IRIS defpattern function. The user must provide as input to the function the first index in the color map to be used, a 64x64 array of type Colorindex to hold the multi-color texture pattern and the name of the file containing the multi-color texture pattern produced by *texted*. The implementation of *defcolorpattern3* is shown in Figure 4.6. The pattern size and the number of colors used in the multi-color texture pattern are read from the file "filename". The function then repeats a loop as many times

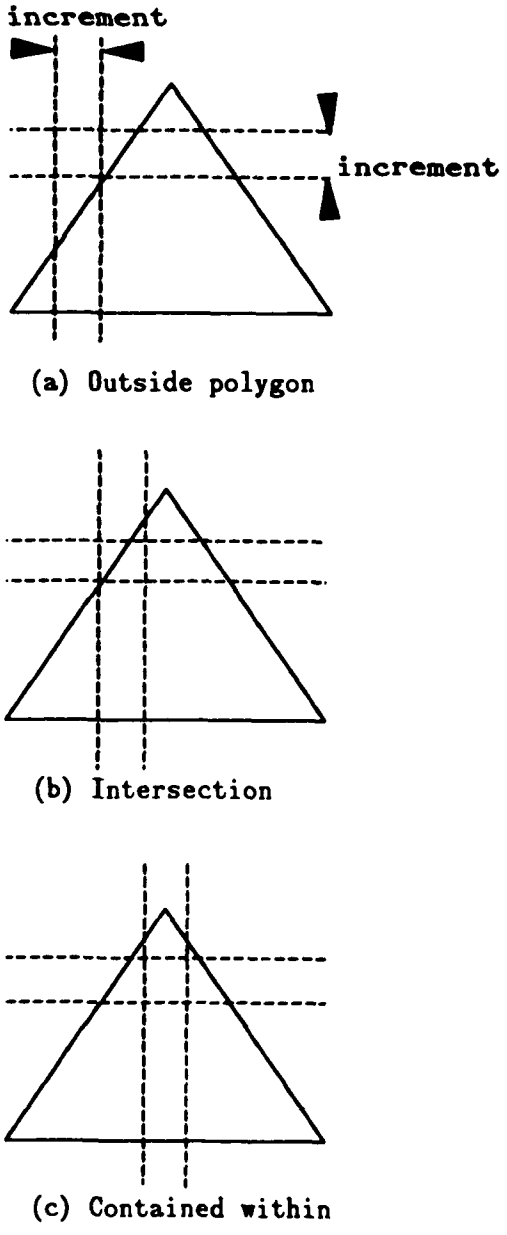


Figure 4.5 Calculation of texture point polygons.

```
defcolorpattern3(first_color,pattern,filename)
Colorindex first_color;
Colorindex pattern[64][64];
char filename[];
{
 short bitword, pattern_size, number_of_colors;
 Colorindex next_color;
 short red_value,green_value,blue_value;
 short i,j,k,m;
 FILE *fp,*fopen();
 fp=fopen(filename,"r");

 /* Read from file pattern size and number of colors used. */
 fscanf(fp,"%hd%hd",&pattern_size,&number_of_colors);

 next_color=first_color;

 /* Repeat for each color used in the multi-color texture pattern. */
 for (k=1; k<=number_of_colors; k++){

  /* Read the RGB values defining the color and place in color map. */
  fscanf(fp,"%hd%hd%hd",&red_value,&green_value,&blue_value);
  mapcolor(next_color,red_value,green_value,blue_value);

  /* Determine indices of "pattern" to hold color map index. */
  for (i=0; i<pattern_size; i++){
   for (j=0; j<pattern_size/16; j++){

    /* Save color map index in array "pattern". */
    fscanf(fp,"%hx",&bitword);
    for (m=0; m<16; m++){
     if ((bitword&0x8000)>0)
      pattern[i][j*16+m]=next_color;
     bitword=bitword<<1;
    }
       }
     }
     next_color+=1;          /* Advance color map index. */
    }
    fclose(fp);
    return(number_of_colors);
   }
```

Figure 4.6 Function to define multi-color texture pattern (3D).

as there are colors in the texture pattern. For each color used in the multi-color texture pattern, the RGB values are read and placed in the color map. The words which before defined a bit mask to be stored in the system table of patterns are now read to determine the indices of the 64x64 array "pattern" that will hold the color map index of the color just defined. A 16x16 or 32x32 multi-color texture pattern only uses the lower indices of the 64x64 array. At the completion of *defcolorpattern3*, the multi-color texture pattern saved in "filename" is defined by a two dimensional array of color map indices. The number of colors used by the multi-color texture pattern is returned so the next available color map index can be determined.

The function

```
polcolorf3(pattern_size,rotate,horz,vert,incr,pattern,n,parray)
short pattern_size;
Angle rotate;
int horz,vert;
Coord incr;
Colorindex pattern[64][64];
long n;
Coord parray[][3];
```

fills a polygon defined in three space with the multi-color texture pattern defined by *defcolorpattern3*. It requires as input a pattern size (16, 32 or 64), a rotation angle "rotate" about the z axis, arguments "horz" and "vert" used to position the texture pattern reference point, the texture point polygon size "incr", and the 64x64 array that defines the multi-color texture pattern, the number of points in the polygon and the polygon coordinates. The implementation of *polcolorf3* is shown in Figure 4.7. The function begins by mapping the polygon defined in three space to the x-y plane. The function *map_to_the_xy_plane* takes as inputs the rotation angle "rotate" about the z axis, the

```
polcolorf3(pattern_size,rotate,horz,vert,incr,pattern,n,parray)
short pattern_size;
Angle rotate;
int horz,vert;
Coord incr;
Colorindex pattern[64][64];
long n;
Coord parray[][3];
{
 Coord xmin,ymin,xmax,ymax,xincr,yincr;
 register int i,j;
 int i_begin,j_begin;
 Matrix trans;

 pushattributes();
 set_up();

 map_to_xy_plane(trans,rotate,n,parray);
 texture_pattern_limits(n,parray,&xmin,&ymin,&xmax,&ymax);

 i_begin=(pattern_size-vert)%pattern_size;
 j_begin=(pattern_size-horz)%pattern_size;
 i=i_begin;    j=j_begin;
 yincr=0.0;
 while ((ymin+yincr)<=ymax){     /* Repeat until polygon partitioned. */
  xincr=0.0;
  while ((xmin+xincr)<=xmax){
   /* A value of 0 in the array "pattern" indicates transparent. */
   if (pattern[i][j]!=0)
     clip_and_fill(trans,pattern[i][j],xmin+xincr,ymin+yincr,
           xmin+xincr+incr,ymin+yincr+incr,n,parray);
   j+=1;
   xincr+=incr;
   if (j==pattern_size) j=0;    /* Check to repeat columns of pattern. */
  }
  j=j_begin;
  i+=1;
  if (i==pattern_size) i=0;      /* Check to repeat rows of pattern. */
  yincr+=incr;
 }
 popattributes();
}
```

Figure 4.7 Function to fill a polygon with multi-color texture pattern.

number of polygon points and the coordinates of the polygon defined in three space. The new polygon defined in the x-y plane is returned with the inverse transformation matrix "trans" to be used in reverse mapping of texture point polygons. The texture pattern limits are determined with the function *texture_pattern_limits*. The lower limit and the arguments "horz" and "vert" are used to calculate the beginning indices to reference the texture pattern array. The upper and lower limit ensure the polygon in the x-y plane is completely partitioned into texture point polygons. The function *clip_and_fill* calculates each texture point polygon and maps it to its proper relative position in three space with the transformation matrix "trans" before filling. Each time the routine *clip_and_fill* is called a new color map index is calculated, as well as new clipping lines. The function *clip_and_fill* takes as inputs the inverse mapping matrix "trans", the color map index from the array "pattern", the clipping lines, the number of points of the polygon and the coordinates of the polygon in the x-y plane.

F. EXAMPLE

A partial example is shown in Figure 4.8 that demonstrates the use of the new 3D multi-color texture pattern functions. The complete program can be found in Appendix C. The program draws a brick pyramid rotating on a checkered plain surface. Each object is filled with a texture pattern produced by *texted*.

After some IRIS initialization and variable definitions, the first multi-color texture pattern is defined with

```
n=defcolorpattern(1,8,"bkgrnd.pat").
```

This defines a multi-color texture pattern starting with the index 1 in the system table of

```
main()
{
   .
   .
   .
 ginit();                    /* Initialize the IRIS. */
   .
   .
   .
 /* Define texture patterns. */
 n=defcolorpattern(1,8,"bkgrnd.pat");
 m=defcolorpattern3(8+n,a_array,"pyr.pat");
 p=defcolorpattern3(8+n+m,b_array,"chex.pat");
   .
   .
 pyr=genobj();
 makeobj(pyr);       /* Make the pyramid. */
  polcolorf3(64,0,-5,0,2.0,a_array,3,face1coords);
  polcolorf3(64,0,-5,0,2.0,a_array,3,face2coords);
  polcolorf3(64,0,-5,0,2.0,a_array,3,face3coords);
  polcolorf3(64,0,-5,0,2.0,a_array,3,face4coords);
 closeobj();

 plain=genobj();
 makeobj(plain);     /* Make the plain surface. */
  polcolorf3(16,0,0,0,25.0,b_array,4,plaincoords);
 closeobj();
   .
   .
 colorclear(1,8,n);     /* Draw background. */
   .
   .
 callobj(plain);        /* Draw plain surface. */
   .
   .
 callobj(pyr);          /* Draw pyramid. */
   .
   .
   .
 gexit();
}
```

Figure 4.8 Example demonstrating texture pattern functions.

patterns and the color map index of 8. The file "bkgrnd.pat" is the multi-color texture pattern that is used for the background. The variable "n" is assigned the value equal to the number of colors used in the multi-color texture pattern.

The function call

m=defcolorpattern3(8+n,a_array,"pyr.pat")

defines the multi-color texture pattern for filling each pyramidal face. It uses the value of "n" to calculate the starting color map index and places the texture pattern of "pyr.pat" in the array a_array. The variable "m" again takes on the value of the number of colors used.

The plain surface texture pattern is defined with

p=defcolorpattern3(8+n+m,b_array,"chex.pat").

The first available color map index is passed in as 8+n+m and the multi-color texture pattern of file "chex.pat" is saved in the array b_array.

The pyramid is made into an object named "pyr" and calls *polcolorf3* for each face of the pyramid to demonstrate the ability of the system to fill a polygon defined anywhere in three space. The first pyramid face is drawn with the function call

polcolorf3(64,0,-5,0,2.0,a_array,3,face1coords).

The texture pattern saved in a_array (read from pyr.pat) is a 64x64 texture pattern. The rotation angle about the z axis used for positioning of the texture pattern is zero. The texture pattern is shifted to the left -5 to make texture pattern edges match where pyramid faces meet. The texture point polygon size in this case is 4 pixels (2.0x2.0) and there are three points in the array "face1coords" which define the pyramid face.

The plain surface is also made into an object named "plain" and is drawn with

polcolorf3(16,0,0,0,25.0,b_array,4,plaincoords).

The texture pattern in b_array (read from chex.pat) is a 16x16 texture pattern and the texture pattern is not shifted. The texture point polygon size in this case is 625 pixels (25.0x25.0) and there are four points in the array "plaincoords" which define the plain surface.

With the multi-color texture patterns defined and the objects built, the screen is cleared by

colorclear(1,8,n).

This texture pattern function makes use of IRIS texture pattern hardware as discussed in Chapter III. The plain surface and the pyramid are then drawn with

callobj(plain);

and

callobj(pyr).

Figure 4.10 is the generated display of this example program.

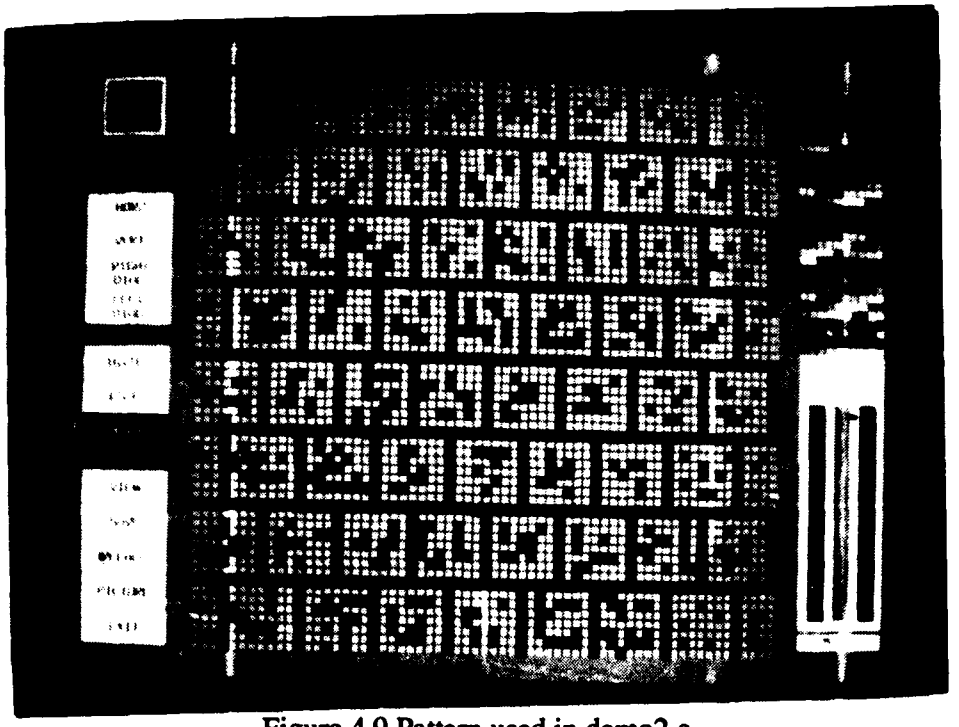


Figure 4.9 Pattern used in demo2.c.

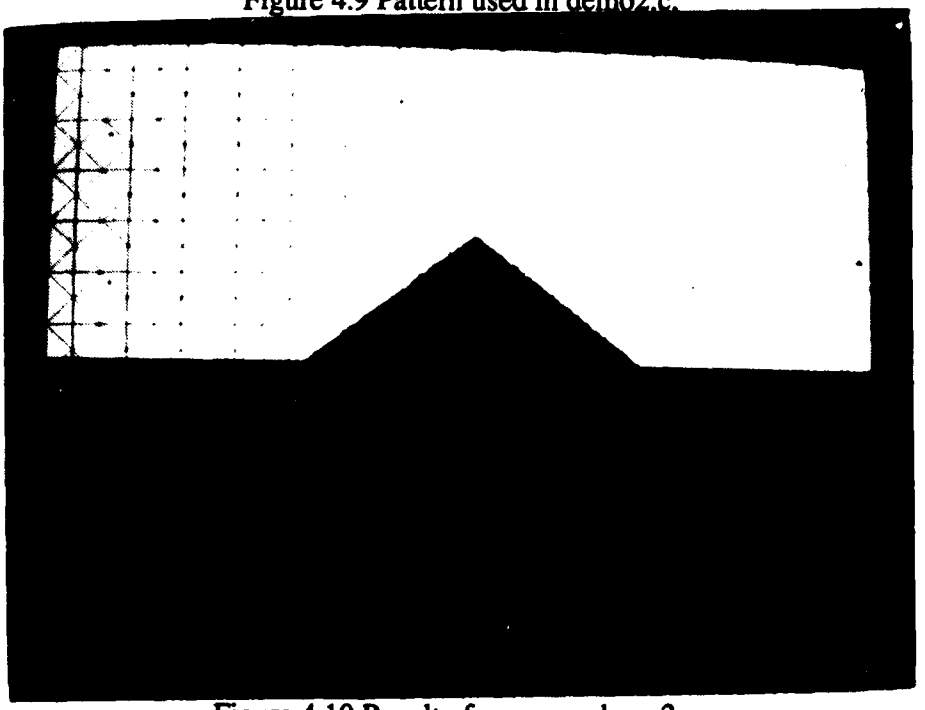


Figure 4.10 Result of program demo2.c.

V. EVALUATION AND PERFORMANCE

A. REAL-TIME TEXTURING

Past research has produced methods to calculate individual pixel intensities based on a mapping from texture space to object space and finally to screen coordinates [1-7]. These same type of methods are used in rendering programs to aid in generating a two-dimensional array of pixel intensities based on a three-dimensional description of a scene [10]. Texturing becomes a part of an overall program to render a realistic image. If this rendering is done on a single processor system, turnaround time is usually measured in hours.

Real-time computer animation implies that we are going to update and display our three-dimensional objects at a rate of approximately 15 updates per second. We accomplish this movement by changing the view position and view direction and then redisplaying our three-dimensional objects. With textures or texture patterns, this implies that individual texture points also change relative to the view position. Individual texture points become larger as objects move closer to the view position and become smaller as objects move away.

B. CHOICE OF METHODS

To approach real-time texturing of objects, our goal had to be to use as much available hardware on the IRIS as possible. After realizing the limitations of the IRIS hardware for texture patterns, the use of polygons to represent each point of a texture pattern seemed natural.

The capabilities of the IRIS are due to the use of a custom VLSI chip called the Geometry Engine. The IRIS makes use of a pipeline of twelve Geometry Engines that accept polygons in user-defined coordinates with rotations, scaling and clipping performed by that pipeline. By using polygons to represent each point of a texture pattern, the Geometry Engines perform the transformation of the object space coordinates of the texture point polygons to screen coordinates. With the Geometry Engines performing transformations, individual texture points become larger as the objects move closer to the view point and smaller as they move away. Hidden surface elimination is achieved by using the IRIS' backface removal hardware. This breaks down for single pixel polygons. In that case, z-buffering is then required for proper textured surface display.

C. FACTORS AFFECTING PERFORMANCE

The performance of the three-dimensional texture pattern functions depends on the number of points required to define the polygon filled with the texture pattern, and the size of the texture point polygons. Increasing the number of points in the polygon results in more comparisons to be made to calculate texture point polygons. Decreasing the texture point polygon size results in increasing the number of coordinates calculated and the number of polygons to fill.

D. QUANTITATIVE DATA

Measurements were taken to calculate the total number of polygons and the amount of CPU time required to generate the display of the example program in Appendix C. There are a total of 19,289 polygons in the display with each pyramidal face

partitioned into 4484 polygons (See Fig. 4.9). It takes approximately 150.35 seconds of CPU time to generate the entire display. This includes the time to build the objects and then display them. Total time to generate the display is reduced by generating only one pyramidal face and using the IRIS rotation capabilities to generate the other three pyramidal faces. Figure 5.1 demonstrates building the pyramid using only one pyramidal face. Using the objects of Figure 5.1, the entire display is generated using approximately 44.57 seconds of CPU time. Additional data was taken varying the texture point polygon size and measuring the time to build the pyramidal object of Figure 5.1 (Table 5.1).

```
/* Make one pyramidal face to generate pyramid. */
face=genobj();
makeobj(face);
  polcolorf3(64,0,-5,0,2.0,a_array,3,a);
closeobj();

/* Make the pyramid object. */
pyr=genobj();
makeobj(pyr);
  callobj(face);        /* face 1 */
  pushmatrix();
  rotate(900,'Y');
  callobj(face);        /* face 2 */
  popmatrix();
  pushmatrix();
  rotate(1800,'Y');
  callobj(face);        /* face 3 */
  popmatrix();
  pushmatrix();
  rotate(-900,'Y');
  callobj(face);        /* face 4 */
  popmatrix();
closeobj();
```

Figure 5.1 Pyramid object built with one face.

By using the IRIS capability to build objects, the overhead associated with calculating texture point polygons can be limited to object construction time. The speed at which an object is updated and displayed with a change in point of view depends solely on the IRIS' capability to transform and fill the polygons that make up the object. Data was taken varying the texture point polygon size and measuring the time to display the pyramid object after the object had already been built (Table 5.2).

Table 5.1 CPU time to build pyramidal object.

increment	#polygons	CPU time(secs.)	polys./sec.
1.0	70,632	121.66667	580
2.0	17,936	30.65000	585
3.0	8092	13.58333	595
4.0	4616	7.85000	588
5.0	3008	5.16667	582
6.0	2116	3.61667	585
7.0	1572	2.68333	585
8.0	1224	2.05000	597
9.0	984	1.66667	590
10.0	808	1.46667	551

Table 5.2 Update time for pyramidal object.

increment	#polygons	Update time(secs.)	polys./sec.
1.0	70,632	4.41667	15,992
2.0	17,936	1.05000	17,082
3.0	8092	0.50000	16,184
4.0	4616	0.30000	15,387
5.0	3008	0.21667	13,883
6.0	2116	0.13333	15,870
7.0	1572	0.11667	13,474
8.0	1224	0.08333	12,240
9.0	984	0.06667	14,760
10.0	808	0.06667	12,119

VI. CONCLUSIONS

A. A COLOR TEXTURE PATTERN EDITOR

Texted proved to be a very valuable tool in designing multi-color texture patterns. The fact that multi-color texture patterns are defined by a number of short integers becomes transparent to the user. The ability to choose or define any color, as well as a selection for transparency, enables the user to design a texture pattern that is limited only by the creativity of the user.

A shortcoming of the editor is the limit of a 64x64 texture pattern as the largest pattern selection. This design decision was based on the fact that the largest texture pattern that can be defined by the IRIS patterning hardware is a 64x64 pixel pattern. Using our 3D texturing technique, there is no reason we should be limited by a 64x64 texture pattern. This limit placed on our editor design made the picture option of little use. Textures such as grass or rocks which might be taken from a photograph do not usually exhibit a repeating pattern. A nice feature would be the ability to select a texture as large as the polygon we wished to fill.

B. REAL-TIME TEXTURING

We pointed out above that texturing using the IRIS' patterning hardware has its limitations. Though the hardware is unable to translate or rotate the developed texture patterns, it can still be very useful in real-time animation. One application has been the use of a grass texture pattern in a road rally simulator. Translation of the grass texture pattern is achieved by using four different grass texture patterns with the translation

occurring from one texture pattern to the other. Though not completely accurate, it is enough to create the illusion to the user who is concentrating on the road anyway.

As can be seen from the data of Chapter V, our software technique for texturing falls short of the 15 updates per second until the texture point polygon increment is 9.0. With an increment of 9.0, each pyramidal face is partitioned into 246 polygons. However, this is not the detail sought when we normally speak of texturing. It is desireable to achieve real-time animation speed updates with a texture point polygon size of 2.0 (17,936 polygons/pyramid) or even 1.0 (70,632 polygons/pyramid). The algorithm for partitioning polygons into texture point polygons is extremely slow, as can be seen from the data. The IRIS as of now still lacks the speed to transform and display the number of polygons required for realistic texturing.

C. FUTURE WORK

With the ability to take texture patterns of any size from photographs, we envision the development of a texture pattern library. Any user developing a computer generated scene would have at his disposal a large number of textures which might include rocks, sand, fields, brick and others. Users could take advantage of the creative abilities of others whose texture patterns could also be saved in this library.

Our functions were developed to be as efficient as possible and yet preserve understandability. There are always new and innovative ways to improve performance. Access to low-level graphics system routines could prove to be a way to improve performance.

We have not even considered the addition of surface roughness or shading, both of which are necessary to achieve a realistic image. Our technique partitioned a polygon

into flat shaded texture point polygons. With faster processors or even multi-processors, our functions could be modified to partition polygons into Gouraud shaded texture point polygons, making even more realistic images. We believe with systems like the IRIS, the ability to use realistic textured objects in real-time computer animation using the technique described is attainable in the near future.

APPENDIX A - COLOR TEXTURE PATTERN FUNCTIONS

defcolorpattern **defcolorpattern**

NAME

defcolorpattern - defines a multi-color texture pattern.

SPECIFICATION

```
C   long defcolorpattern(first_pattern,first_color,filename)
    short first_pattern;
    Colorindex first_color;
    char filename[];
```

DESCRIPTION

defcolorpattern defines an arbitrary multi-color texture pattern made up of an arbitrary number of texture patterns saved in the system table of patterns. The first argument specifies the beginning index in the system table of patterns where the multi-color texture pattern will be stored. The next argument is the beginning index of the color map where the colors of the multi-color texture pattern will be stored. Finally, the last argument is the filename of the texture pattern created through texted. The function returns the number of colors used in the texture pattern, which corresponds to the number of *indices* used in the system table of patterns as well as the number of color map entries.

NOTE

This command can be used only in immediate mode.

NAME

rectcolorf - fills a rectangular area with a multi-color texture pattern defined by defcolorpattern.

SPECIFICATION

```
C   rectcolorf(first_pattern,first_color,number_of_colors,x1,y1,x2,y2)
    short first_pattern;
    Colorindex first_color;
    short number_of_colors;
    Coord x1,y1,x2,y2;

    rectcolorfi(first_pattern,first_color,number_of_colors,x1,y1,x2,y2)
    short first_pattern;
    Colorindex first_color;
    short number_of_colors;
    Icoord x1,y1,x2,y2;

    rectcolorfs(first_pattern,first_color,number_of_colors,x1,y1,x2,y2)
    short first_pattern;
    Colorindex first_color;
    short number_of_colors;
    Scoord x1,y1,x2,y2;
```

DESCRIPTION

rectcolorf produces a filled rectangular region, using the multi-color texture pattern defined by defcolorpattern. The first argument is the first index in the system table of patterns which the multi-color texture pattern uses. The second argument is the starting color map index of the colors used in the multi-color texture pattern. The third argument is the number of colors used in the multi-color texture pattern, which also corresponds to the number of indices used in the system table of patterns. Since a rectangle is a two-dimensional shape, rectcolorf takes only 2D arguments, and sets the z coordinates to zero. The points (x1,y1) and (x2,y2) are the opposite corners of the rectangle. The current graphics position is set to (x1,y1) after the region is drawn.

NAME

polcolorf - draws a filled polygon with a multi-color texture pattern defined by defcolorpattern.

SPECIFICATION

```
C   polcolorf(first_pattern,first_color,number_of_colors,n,parray)
    short first_pattern;
    Colorindex first_color;
    short number_of_colors;
    long n;
    Coord parray[][3];

    polcolorfi(first_pattern,first_color,number_of_colors,n,parray)
    short first_pattern;
    Colorindex first_color;
    short number_of_colors;
    long n;
    Icoord parray[][3];

    polcolorfs(first_pattern,first_color,number_of_colors,n,parray)
    short first_pattern;
    Colorindex first_color;
    short number_of_colors;
    long n;
    Scoord parray[][3];

    polcolorf2(first_pattern,first_color,number_of_colors,n,parray)
    short first_pattern;
    Colorindex first_color;
    short number_of_colors;
    long n;
    Coord parray[][2];

    polcolorf2i(first_pattern,first_color,number_of_colors,n,parray)
    short first_pattern;
    Colorindex first_color;
    short number_of_colors;
    long n;
    Icoord parray[][2];
```

```
polcolorf2s(first_pattern,first_color,number_of_colors,n,parray)
short first_pattern;
Colorindex first_color;
short number_of_colors;
long n;
Scoord parray[][2];
```

DESCRIPTION

polcolorf fills polygonal areas, using the multi-color texture pattern defined by defcolorpattern. The first argument is the first index in the system table of patterns which the multi-color texture pattern uses. The second argument is the starting color map index of the colors used in the multi-color texture pattern. The third argument is the number of colors used in the multi-color texture pattern, which also corresponds to the number of indices used in the system table of patterns. The final two arguments are the number of points in the polygon and the array of points which define the polygon. The points can be expressed as integers, shorts, or floating point, in 2D or 3D space. Two-dimensional polygons are drawn with z=0. After the polygon is filled, the current graphics position is set to the first point in the array.

The results are undefined if the polygon is not convex.

NAME

circcolorf - draws a filled circle with a multi-color texture pattern defined by defcolorpattern.

SPECIFICATION

```
C   circcolorf(first_pattern,first_color,number_of_colors,x,y,radius)
    short first_pattern;
    Colorindex first_color;
    short number_of_colors;
    Coord x,y,radius;

    circcolorfi(first_pattern,first_color,number_of_colors,x,y,radius)
    short first_pattern;
    Colorindex first_color;
    short number_of_colors;
    Icoord x,y,radius;

    circcolorfs(first_pattern,first_color,number_of_colors,x,y,radius)
    short first_pattern;
    Colorindex first_color;
    short number_of_colors;
    Scoord x,y,radius;
```

DESCRIPTION

circcolorf fills a circle, using the multi-color texture pattern defined by defcolorpattern. The first argument is the first index in the system table of patterns which the multi-color texture pattern uses. The second argument is the starting color map index of the colors used in the multi-color texture pattern. The third argument is the number of colors used in the multi-color texture pattern, which also corresponds to the number of indices used in the system table of patterns. The circle has its center at (x,y) and a radius radius, both specified in world coordinates. Since a circle is a two-dimensional shape, these commands have only 2D forms. The circle is drawn in the x-y plane, with z=0.

NAME

arccolorf - fills a circular arc with a multi-color texture pattern defined by defcolorpattern.

SPECIFICATION

```
C   arccolorf(first_pattern,first_color,number_of_colors,x,y,radius,
        startang,endang)
    short first_pattern;
    Colorindex first_color;
    short number_of_colors;
    Coord x,y,radius;
    Angle startang,endang;

    arccolorfi(first_pattern,first_color,number_of_colors,x,y,radius,
        startang,endang)
    short first_pattern;
    Colorindex first_color;
    short number_of_colors;
    Icoord x,y,radius;
    Angle startang,endang;

    arccolorfs(first_pattern,first_color,number_of_colors,x,y,radius,
        startang,endang)
    short first_pattern;
    Colorindex first_color;
    short number_of_colors;
    Scoord x,y,radius;
    Angle startang,endang;
```

DESCRIPTION

arccolorf draws a filled circular arc, using the multi-color texture pattern defined by defcolorpattern. The first argument is the first index in the system table of patterns which the multi-color texture pattern uses. The second argument is the starting color map index of the colors used in the multi-color texture pattern. The third argument is the number of colors used in the multi-color texture pattern, which also corresponds to the number of indices used in the system table of patterns. The arc is specified as a center point, a starting angle, an ending angle, and a radius. The angle is measured from the x-axis and specified in integral tenths of degrees. Positive angles describe counterclockwise rotations. Since an arc is a two-dimensional shape, these commands have only 2D forms. The arc is in the

x-y plane with z=0. Arcs are drawn counterclockwise from startang to endang, so the arc from 10 degrees to 5 degrees is a nearly complete circle.

NAME

colorclear - clears the viewport with a multi-color texture pattern defined by defcolorpattern.

SPECIFICATION

```
C   colorclear(first_pattern,first_color,number_of_colors)
    short first_pattern;
    Colorindex first_color;
    short number_of_colors;
```

DESCRIPTION

colorclear clears the screen area within the current viewport, using the multi-color texture pattern defined by defcolorpattern. The first argument is the first index in the system table of patterns which the multi-color pattern uses. The second argument is the starting color map index of the colors used in the multi-color texture pattern. The third argument is the number of colors used in the multi-color texture pattern, which also corresponds to the number of indices used in the system table of patterns.

NAME

defcolorpattern3 - defines a multi-color texture pattern and stores it in the form of an array.

SPECIFICATION

C long defcolorpattern3(first_color,pattern,filename)
Colorindex first_color;
Colorindex pattern[64][64];
char filename[];

DESCRIPTION

defcolorpattern3 defines an arbitrary multi-color texture pattern and saves it in the form of a 64x64 array. The first argument specifies the beginning index in the color map where the colors of the multi-color texture pattern will be stored. The second argument is the 64x64 array that will hold the multi-color texture pattern after defcolorpattern3 is called. Finally, the last argument is the filename of the texture pattern created through texted. The function returns the number of colors used in the multi-color texture pattern, which corresponds to the number of color map entries made.

NOTE

This command can be used only in immediate mode.

NAME

polcolorf3 - draws a filled polygon with a multi-color texture pattern defined by defcolorpattern3.

SPECIFICATION

```
C   polcolorf3(pattern_size,rotate,horz,vert,incr,pattern,n,parray)
    short pattern_size;
    Angle rotate;
    int horz,vert;
    Coord incr;
    Colorindex pattern[64][64];
    long n;
    Coord parray[][3];

    polcolorf3i(pattern_size,rotate,horz,vert,incr,pattern,n,parray)
    short pattern_size;
    Angle rotate;
    int horz,vert;
    Icoord incr;
    Colorindex pattern[64][64];
    long n;
    Icoord parray[64][64];

    polcolorf3s(pattern_size,rotate,horz,vert,incr,pattern,n,parray)
    short pattern_size;
    Angle rotate;
    int horz,vert;
    Scoord incr;
    Colorindex pattern;
    long n;
    Scoord parray[64][64];
```

DESCRIPTION

polcolorf3 fills polygonal areas, using the multi-color texture pattern defined by defcolorpattern3. The first argument is the texture pattern size which can take on values of 16(16x16), 32(32x32), or 64(64x64). The next three arguments of "rotate", "horz", and "vert" are used to position the multi-color texture pattern on the polygon surface. The polygon is filled with the texture pattern by first bringing it into the x-y plane and then rotating the polygon about the z axis with the first point of the polygon at the origin, and the second point initially on the x axis.

The angle "rotate" is measured from the x axis and specified in integral tenths of degrees. Positive angles describe counterclockwise rotations. The arguments of "horz" and "vert" shift the texture pattern right or left, up or down. With "horz" and "vert" having values of zero, the reference point is the point defined by the smallest x and y values of all the array points, after the polygon has been positioned in the x-y plane. The argument "incr" specifies the rectangular polygon size which will represent each texture point. The argument "pattern" is the array formed upon an earlier call on defcolorpattern3. The final two arguments are the number of points in the polygon and the array of points which define the polygon. The points can be expressed as integers, shorts, or floating point numbers.

APPENDIX B - EXAMPLE 1

```
/*****************************************************************************
 demo1.c is a program which demonstrates the use of the color texture
 pattern functions which make use of the files produced by texted.
 These particular functions make use of the IRIS hardware to produce
 multi-color texture filled objects.
*****************************************************************************/

#include "gl.h"
#include "device.h"

main()
{
   Colorindex wmask;
   short n,m,p,q;
   Icoord a[6][2];

   /* Coordinates for a 6 sided polygon. */
   a[0][0]=550; a[0][1]=50;
   a[1][0]=450; a[1][1]=150;
   a[2][0]=550; a[2][1]=250;
   a[3][0]=650; a[3][1]=250;
   a[4][0]=750; a[4][1]=150;
   a[5][0]=650; a[5][1]=50;

   /* Initialize the IRIS */
   ginit();

   /* Configure IRIS for single buffer */
   singlebuffer();
   gconfig();

   /* Use all bit planes */
   wmask=((1<<getplanes())-1);
   writemask(wmask);

   /* Use full screen */
   viewport(0,1023,0,767);
   ortho2(0.0,1023.0,0.0,767.0);

   cursoff();

   /* The color texture pattern saved in "brick.out" is stored in the
      system table of patterns starting at index 1, and in the color
      map starting at index 8. "n" will equal the number of colors
      used in the color texture pattern, which also corresponds to the
      number of indices used in the system table of patterns. */

   n=defcolorpattern(1,8,"brick.pat");
```

```
/* The color texture pattern saved in "wood.pat" is stored in the
   system table of patterns starting at index 1+n, and in the color
   map starting at index 8+n. "m" will equal the number of colors
   used in the color texture pattern, which also corresponds to the
   number of indices used in the system table of patterns. */

m=defcolorpattern(1+n,8+n,"wood.pat");

/* The color texture pattern saved in "weave.pat" is stored in the
   system table of patterns starting at index 1+n+m, and in the color
   map starting at index 8+n+m. "p" will equal the number of colors
   used in the color texture pattern, which also corresponds to the
   number of indices used in the system table of patterns. */

p=defcolorpattern(1+n+m,8+n+m,"weave.pat");

/* The color texture pattern saved in "mod.pat" is stored in the
   system table of patterns starting at index 1+n+m+p, and in the color
   map starting at index 8+m+n+p. "q" will equal the number of colors
   used in the color texture pattern, which also corresponds to the
   number of indices used in the system table of patterns. */

q=defcolorpattern(1+m+n+p,8+m+n+p,"mod.pat");

/* Background of white */
color(WHITE);
clear();

/* Fill a rectangle with the texture patterns that start at index 1
   in the system table of patterns, index 8 in the color map, and
   uses a total of "n" indices. */

rectcolorfi(1,8,n,75,500,425,700);

/* Fill a circle with the texture patterns that start at index 1+n
   in the system table of patterns, index 8+n in the color map,
   and uses a total of "m" indices. */

circcolorfi(1+n,8+n,m,250,250,100);

/* Fill an arc with the texture patterns that start at index 1+n+m
   in the system table of pattterns, index 8+n+m in the color map,
   and uses a total of "p" indices. */

arccolorfi(1+n+m,8+n+m,p,550,500,200,300,900);

/* Fill a polygon  with the texture patterns that start at index
  1+n+m+p in the system table of patterns, 8+n+m+p in the color map,
  and uses a total of "q" indices. */

polcolorf2i(1+n+m+p,8+n+m+p,q,6,a);
```

```
    /* Depress RIGHTMOUSE to exit program */
    while(TRUE){
     if (getbutton(MOUSE1)==1)
        break;
    }

    /* Restore IRIS */
    color(BLACK);
    clear();
    curson();
    gexit();
}
```

APPENDIX C - EXAMPLE 2

```
/*****************************************************************************
 demo2.c is a program which demonstrates the use of the color texture
 pattern functions that use the files produced by texted. The functions
 which make use of the IRIS hardware as well as those that give the user
 the capability to manipulate those texture patterns in three space are
 demonstrated.
*****************************************************************************/

#include "gl.h"
#include "device.h"

main()
{
   Colorindex wmask;
   Coord face1coords[3][3],face2coords[3][3];
   Coord face3coords[3][3],face4coords[3][3],plaincoords[4][3];
   register i,j,num;
   Colorindex a_array[64][64];
   Colorindex b_array[64][64];
   Object pyr,plain;
   short rot=0;
   int n,m,p;

/* Coordinates of the four faces of the pyramid. */
  face1coords[0][0]=100.0; face1coords[0][1]=0.0;   face1coords[0][2]=100.0;
  face1coords[1][0]=100.0; face1coords[1][1]=0.0;   face1coords[1][2]=(-100.0);
  face1coords[2][0]=0.0;   face1coords[2][1]=142.2; face1coords[2][2]=0.0;

  face2coords[0][0]=100.0; face2coords[0][1]=0.0;   face2coords[0][2]=(-100.0);
  face2coords[1][0]=(-100.0); face2coords[1][1]=0.0; face2coords[1][2]=(-100.0);
  face2coords[2][0]=0.0;   face2coords[2][1]=142.2; face2coords[2][2]=0.0;

  face3coords[0][0]=(-100.0); face3coords[0][1]=0.0; face3coords[0][2]=100.0;
  face3coords[1][0]= 100.0;   face3coords[1][1]=0.0; face3coords[1][2]=100.0;
  face3coords[2][0]=0.0;      face3coords[2][1]=142.2; face3coords[2][2]=0.0;

  face4coords[0][0]=(-100.0); face4coords[0][1]=0.0; face4coords[0][2]=(-100.0);
  face4coords[1][0]=(-100.0); face4coords[1][1]=0.0; face4coords[1][2]=100.0;
  face4coords[2][0]=0.0;      face4coords[2][1]=142.2; face4coords[2][2]=0.0;

/* Coordinates of the plain surface. */
  plaincoords[0][0]=(-500.0); plaincoords[0][1]=0.0; plaincoords[0][2]=0.0;
  plaincoords[1][0]=500.0;    plaincoords[1][1]=0.0; plaincoords[1][2]=0.0;
  plaincoords[2][0]=500.0;    plaincoords[2][1]=0.0; plaincoords[2][2]=(-800.0);
  plaincoords[3][0]=(-500.0); plaincoords[3][1]=0.0; plaincoords[3][2]=(-800.0);

   /* Initialize IRIS */
   ginit();

   /* Configure for double buffer */
   doublebuffer();
   gconfig();
```

```
/* Use backface polygon removal for hidden surface elimination. */
backface(TRUE);

/* Use all bit planes */
wmask=((1<<getplanes())-1);
writemask(wmask);

/* Use full screen */
viewport(0,1023,0,767);
perspective(550,1.0,0.0,1000.0);
lookat(0.0,100.0,-5.0,0.0,0.0,-900.0,0);

cursoff();

/* The color texture pattern saved in "bkgrnd.pat" is stored in the
   system table of patterns starting at index 1, and in the color
   map starting at index 8. "n" will equal the number of colors
   used in the color texture pattern, which also corresponds to the
   number of indices used in the system table of patterns. */

n=defcolorpattern(1,8,"bkgrnd.pat");

/* The color texture pattern saved in "pyr.pat" is stored in the
   color map starting at index 8+n, and in a_array as an array of
   color map indices. */

m=defcolorpattern3(8+n,a_array,"pyr.pat");

/* The color texture pattern saved in "chex.pat" is stored in the
   color map starting at index 8+n+m, and in b_array as an array of
   color map indices. */

p=defcolorpattern3(8+n+m,b_array,"chex.pat");

/* Build the pyramid object. Each face of the pyramid uses a 64x64
   color texture pattern saved in a_array. The color texture pattern
   reference point for each face is shifted left 5. There is no
   positioning of the color texture pattern reference point vertically,
   or rotation about the first point of the polygon. The size chosen
   for a texture point is 2.0. */

pyr=genobj();
makeobj(pyr);
  polcolorf3(64,0,-5,0,2.0,a_array,3,face1coords);
  polcolorf3(64,0,-5,0,2.0,a_array,3,face2coords);
  polcolorf3(64,0,-5,0,2.0,a_array,3,face3coords);
  polcolorf3(64,0,-5,0,2.0,a_array,3,face4coords);
closeobj();
```

```
    /* Build the plain object. The plain object uses a 16x16 color texture
       pattern saved in b_array. There is no positioning of the color
       texture pattern and the size chosen for a texture point is 25.0. */

    plain=genobj();
    makeobj(plain);
     polcolorf3(16,0,0,0,25.0,b_array,4,plaincoords);
    closeobj();

    while(TRUE){

     /* Clear screen to the texture patterns that start at index 1 in
        the system table of patterns, index 8 in the color map, and
        uses a total of "n" indices. */

     colorclear(1,8,n);

     /* Draw the plain surface. */
     callobj(plain);

     /* After pyramid has rotated 360 degrees, initialize rotation
        angle to 0 */

     if (rot==3600)
              rot=0;

     /* Translate and rotate the pyramid. */
     pushmatrix();
     translate(0.0,0.0,-500.0);
     rotate(rot,'Y');
     callobj(pyr);
     popmatrix();
     swapbuffers();

     /* Increment the rotation angle 10 degrees. */
     rot+=100;
     if (getbutton(MOUSE1)==1)
       break;
    }

    /* Restore IRIS */
    color(BLACK);
    clear();
    curson();
    gexit();
}
```

APPENDIX D - SOURCE CODE FOR COLOR TEXTURE PATTERN FUNCTIONS: 2D

```
/*****************************************************************************
 texture.c contains all the necessary functions to create multi-color
 texture filled objects using the IRIS hardware texture pattern
 capabilities.
 *****************************************************************************/

#include "gl.h"
#include "device.h"
#include <stdio.h>

/*****************************************************************************
 ROUTINE:      defcolorpattern
 *****************************************************************************
 The function defcolorpattern allows the user to define a multi-color
 texture pattern through use of a file produced by texted. The multi-color
 texture pattern is stored as a number of texture patterns in the system
 table of patterns starting with "first_pattern", and as a number of indices
 in the color map starting with "first_color". The function returns the
 number of colors used in the multi-color texture pattern, which also
 corresponds to the number of indices used in the system table of patterns.
 *****************************************************************************/

defcolorpattern(first_pattern,first_color,filename)
short first_pattern;    /* System table of patterns index of first pattern*/
Colorindex first_color; /* Color map index of first color used.*/
char filename[];        /* Filename of saved texture pattern.*/
{
 short number_of_colors;
 short next_pattern;
 Colorindex next_color;
 short pattern_size;
 short red_value,green_value,blue_value;
 short bitword;
 short mask[256];
 short i,k;
 FILE *fp,*fopen();

 /* Open the file and read the pattern size and the number of colors used.*/
 fp=fopen(filename,"r");
 fscanf(fp,"%hd%hd",&pattern_size,&number_of_colors);

 next_pattern=first_pattern;
 next_color=first_color;

 /* For each color used in the multi-color texture pattern. */
 for (k=1; k<=number_of_colors; k++){

  /* Read the RGB values and place in color map. */
  fscanf(fp,"%hd%hd%hd",&red_value,&green_value,&blue_value);
  mapcolor(next_color,red_value,green_value,blue_value);
```

```
  /* Advance to next index in color map. */
  next_color+=1;

  /* Read the words that define the texture pattern. */
  for (i=0; i<(pattern_size*(pattern_size/16)); i++){
    fscanf(fp,"%hx",&bitword);
    mask[i]=bitword;
  }
  /* Save the texture pattern in the system table of patterns. */
  defpattern(next_pattern,pattern_size,mask);

  /* Advance to next index in system table of patterns. */
  next_pattern+=1;
 }
 fclose(fp);
 return(number_of_colors);
}
```

```
/*****************************************************************************
 ROUTINES:      rectcolorfi, rectcolorfs, rectcolorf()
 *****************************************************************************
 The functions rectcolorfi, rectcolorfs, and rectcolorf will fill a
 rectangle with the texture patterns starting with "first_pattern" in
 the system table of patterns and using colors starting with color map
 index "first_color". The argument "number_of_colors" should take on
 the value returned by defcolorpattern.
 *****************************************************************************/

rectcolorfi(first_pattern,first_color,number_of_colors,x1,y1,x2,y2)
short first_pattern;
Colorindex first_color;
short number_of_colors;
Icoord x1,y1,x2,y2;
{
 short i;
 short next_pattern;
 Colorindex next_color;
 pushattributes();
 next_pattern=first_pattern;
 next_color=first_color;
 for (i=1; i<=number_of_colors; i++){
  setpattern(next_pattern);
  next_pattern+=1;
  color(next_color);
  next_color+=1;
  rectfi(x1,y1,x2,y2);
 }
 popattributes();
}

rectcolorfs(first_pattern,first_color,number_of_colors,x1,y1,x2,y2)
short first_pattern;
Colorindex first_color;
short number_of_colors;
Scoord x1,y1,x2,y2;
{
 short i;
 short next_pattern;
 Colorindex next_color;
 pushattributes();
 next_pattern=first_pattern;
 next_color=first_color;
 for (i=1; i<=number_of_colors; i++){
  setpattern(next_pattern);
  next_pattern+=1;
  color(next_color);
  next_color+=1;
  rectfs(x1,y1,x2,y2);
 }
 popattributes();
}
```

```
rectcolorf(first_pattern,first_color,number_of_colors,x1,y1,x2,y2)
short first_pattern;
Colorindex first_color;
short number_of_colors;
Coord x1,y1,x2,y2;
{
 short i;
 short next_pattern;
 Colorindex next_color;
 pushattributes();
 next_pattern=first_pattern;
 next_color=first_color;
 for (i=1; i<=number_of_colors; i++){
  setpattern(next_pattern);
  next_pattern+=1;
  color(next_color);
  next_color+=1;
  rectf(x1,y1,x2,y2);
 }
 popattributes();
}
```

```
/*****************************************************************************
 ROUTINES:      polcolorf2i, polcolorf2s, polcolorf2,
                polcolorfi, polcolorfs, polcolorf
 *****************************************************************************
 The functions polcolorf2i, polcolorf2s, polcolorf2, polcolorfi, polcolorfs,
 and polcolorf will fill a polygon with the texture patterns starting with
 "first_pattern" in the system table of patterns and using colors starting
 with color map index "first_color". The argument "number_of_colors" should
 take on the value returned by defcolorpattern.
 *****************************************************************************/

polcolorf2i(first_pattern,first_color,number_of_colors,n,parray)
short first_pattern;
Colorindex first_color;
short number_of_colors;
long n;
Icoord parray[][2];
{
 short i;
 short next_pattern;
 Colorindex next_color;
 pushattributes();
 next_pattern=first_pattern;
 next_color=first_color;
 for (i=1; i<=number_of_colors; i++){
  setpattern(next_pattern);
  next_pattern+=1;
  color(next_color);
  next_color+=1;
  polf2i(n,parray);
 }
 popattributes();
}

polcolorf2s(first_pattern,first_color,number_of_colors,n,parray)
short first_pattern;
Colorindex first_color;
short number_of_colors;
long n;
Scoord parray[][2];
{
 short i;
 short next_pattern;
 Colorindex next_color;
 pushattributes();
 next_pattern=first_pattern;
 next_color=first_color;
 for (i=1; i<=number_of_colors; i++){
  setpattern(next_pattern);
  next_pattern+=1;
  color(next_color);
  next_color+=1;
  polf2s(n,parray);
```

```
 }
 popattributes();
}

polcolorfi(first_pattern,first_color,number_of_colors,n,parray)
short first_pattern;
Colorindex first_color;
short number_of_colors;
long n;
Icoord parray[][3];
{
 short i;
 short next_pattern;
 Colorindex next_color;
 pushattributes();
 next_pattern=first_pattern;
 next_color=first_color;
 for (i=1; i<=number_of_colors; i++){
  setpattern(next_pattern);
  next_pattern+=1;
  color(next_color);
  next_color+=1;
  polfi(n,parray);
 }
 popattributes();
}

polcolorfs(first_pattern,first_color,number_of_colors,n,parray)
short first_pattern;
Colorindex first_color;
short number_of_colors;
long n;
Scoord parray[][3];
{
 short i;
 short next_pattern;
 Colorindex next_color;
 pushattributes();
 next_pattern=first_pattern;
 next_color=first_color;
 for (i=1; i<=number_of_colors; i++){
  setpattern(next_pattern);
  next_pattern+=1;
  color(next_color);
  next_color+=1;
  polfs(n,parray);
 }
 popattributes();
}
```

```
polcolorf2(first_pattern,first_color,number_of_colors,n,parray)
short first_pattern;
Colorindex first_color;
short number_of_colors;
long n;
Coord parray[][2];
{
 short i;
 short next_pattern;
 Colorindex next_color;
 pushattributes();
 next_patttern=first_pattern;
 next_color=first_color;
 for (i=1; i<=number_of_colors; i++){
  setpattern(next_pattern);
  next_pattern+=1;
  color(next_color);
  next_color+=1;
  polf2(n,parray);
 }
 popattributes();
}

polcolorf(first_pattern,first_color,number_of_colors,n,parray)
short first_pattern;
Colorindex first_color;
short number_of_colors;
long n;
Coord parray[][3];
{
 short i;
 short next_pattern;
 Colorindex next_color;
 pushattributes();
 next_pattern=first_pattern;
 next_color=first_color;
 for (i=1; i<=number_of_colors; i++){
  setpattern(next_pattern);
  next_pattern+=1;
  color(next_color);
  next_color+=1;
  polf(n,parray);
 }
 popattributes();
}
```

```
/*****************************************************************************
 ROUTINES:       circcolorfi, circcolorfs, circcolorf
 *****************************************************************************
 The functions circcolorfi, circcolorfs,and circcolorf will fill a circle
 with the texture patterns starting with "first_pattern" in the system
 table of patterns and using colors starting with color map index
 "first_color". The argument "number_of_colors" should take on the value
 returned by defcolorpattern.
 *****************************************************************************/

circcolorfi(first_pattern,first_color,number_of_colors,x,y,radius)
short first_pattern;
Colorindex first_color;
short number_of_colors;
Icoord x,y,radius;
{
 short i;
 short next_pattern;
 Colorindex next_color;
 pushattributes();
 next_pattern=first_pattern;
 next_color=first_color;
 for (i=1; i<=number_of_colors; i++){
  setpattern(next_pattern);
  next_pattern+=1;
  color(next_color);
  next_color+=1;
  circfi(x,y,radius);
 }
 popattributes();
}

circcolorfs(first_pattern,first_color,number_of_colors,x,y,radius)
short first_pattern;
Colorindex first_color;
short number_of_colors;
Scoord x,y,radius;
{
 short i;
 short next_pattern;
 Colorindex next_color;
 pushattributes();
 next_pattern=first_pattern;
 next_color=first_color;
 for (i=1; i<=number_of_colors; i++){
  setpattern(next_pattern);
  next_pattern+=1;
  color(next_color);
  next_color+=1;
  circfs(x,y,radius);
 }
 popattributes();
}
```

```
circcolorf(first_pattern,first_color,number_of_colors,x,y,radius)
short first_pattern;
Colorindex first_color;
short number_of_colors;
Coord x,y,radius;
{
 short i;
 short next_pattern;
 Colorindex next_color;
 pushattributes();
 next_pattern=first_pattern;
 next_color=first_color;
 for (i=1; i<=number_of_colors; i++){
  setpattern(next_pattern);
  next_pattern+=1;
  color(next_color);
  next_color+=1;
  circf(x,y,radius);
 }
 popattributes();
}
```

```
/*****************************************************************************
 ROUTINES:      arccolorfi, arccolorfs, arccolorf
 *****************************************************************************
 The functions arccolorfi, arccolorfs, and arccolorf will fill
 an arc with the texture patterns starting with "first_pattern"
 in the system table of patterns and using colors starting with
 color map index "first_color". The argument "number_of_colors"
 should take on the value returned by defcolorpattern.
 *****************************************************************************/

arccolorfi(first_pattern,first_color,number_of_colors,x,y,radius,startang,endang)
short first_pattern;
Colorindex first_color;
short number_of_colors;
Icoord x,y,radius;
Angle startang,endang;
{
 short i;
 short next_pattern;
 Colorindex next_color;
 pushattributes();
 next_pattern=first_pattern;
 next_color=first_color;
 for (i=1; i<=number_of_colors; i++){
  setpattern(next_pattern);
  next_pattern+=1;
  color(next_color);
  next_color+=1;
  arcfi(x,y,radius,startang,endang);
 }
 popattributes();
}

arccolorfs(first_pattern,first_color,number_of_colors,x,y,radius,startang,endang)
short first_pattern;
Colorindex first_color;
short number_of_colors;
Scoord x,y,radius;
Angle startang,endang;
{
 short i;
 short next_pattern;
 Colorindex next_color;
 pushattributes();
 next_pattern=first_pattern;
 next_color=first_color;
 for (i=1; i<=number_of_colors; i++){
  setpattern(next_pattern);
  next_pattern+=1;
  color(next_color);
  next_color+=1;
  arcfs(x,y,radius,startang,endang);
```

```
 }
 popattributes();
}

arccolorf(first_pattern,first_color,number_of_colors,x,y,radius,startang,endang)
short first_pattern;
Colorindex first_color;
short number_of_colors;
Coord x,y,radius;
Angle startang,endang;
{
 short i;
 short next_pattern;
 Colorindex next_color;
 pushattributes();
 next_pattern=first_pattern;
 next_color=first_color;
 for (i=1; i<=number_of_colors; i++){
  setpattern(next_pattern);
  next_pattern+=1;
  color(next_color);
  next_color+=1;
  arcf(x,y,radius,startang,endang);
 }
 popattributes();
}
```

```
/*****************************************************************************
 ROUTINE:      colorclear
 *****************************************************************************
 The function colorclear will fill the entire screen with the texture
 patterns starting with "first_pattern" in the system table of patterns
 and using colors starting with color map index "first_color". The
 argument "number_of_colors" should take on the value returned by
 defcolorpatttern.
 *****************************************************************************/

colorclear(first_pattern,first_color,number_of_colors)
short first_pattern;
Colorindex first_color;
short number_of_colors;
{
 short i;
 short next_pattern;
 Colorindex next_color;
 pushattributes();
 next_patternn=first_pattern;
 next_color=first_color;
 for (i=1; i<=number_of_colors; i++){
  setpattern(next_pattern);
  next_pattern+=1;
  color(next_color);
  next_color+=1;
  clear();
 }
 popattributes();
}
```

APPENDIX E - SOURCE CODE FOR COLOR TEXTURE PATTERN FUNCTIONS: 3D

```
/*****************************************************************************
 texture3.c contains all the functions necessary to create multi-color
 texture filled polygons that can be manipulated in three space. This is
 achieved by filling polygons with polygons that represent each texture
 point of a multi-color texture pattern.
 *****************************************************************************/

#include "gl.h"
#include <math.h>
#include <stdio.h>

#define NIL 0

/* Data structure to hold coordinates of a texture point polygon. */
struct linked_list{
   Coord xvertex;
   Coord yvertex;
   Coord zvertex;
   struct linked_list *next;
};
typedef struct linked_list vertices;
typedef vertices *link;

/*****************************************************************************
 ROUTINE:      defcolorpattern3
 *****************************************************************************
 The function defcolorpattern3 allows the user to define a multi-color
 texture pattern through use of a file produced by texted. The multi-color
 texture pattern is stored as an array of color map indices which define
 the pattern. The function returns the number of colors used in the
 multi-color texture pattern.
 *****************************************************************************/

defcolorpattern3(first_color,pattern,filename)
Colorindex first_color;       /* Color map index of first color used. */
Colorindex pattern[64][64];   /* Array to hold texture pattern */
char filename[];              /* File name of texture pattern */
{
 short bitword;
 Colorindex next_color;
 short number_of_colors;
 short red_value,green_value,blue_value;
 short i,j,k,m;
 FILE *fp,*fopen();
 short pattern_size;

 next_color=first_color;

 fp=fopen(filename,"r");
```

```
    /* Read size of the texture pattern and number of colors used. */
    fscanf(fp,"%hd%hd",&pattern_size,&number_of_colors);

    /* Repeat for each color in the multi-color texture pattern. */
    for (k=1; k<=number_of_colors; k++){

     /* Read the rgb values and place in color map. */
     fscanf(fp,"%hd%hd%hd",&red_value,&green_value,&blue_value);
     mapcolor(next_color,red_value,green_value,blue_value);

     /* Save the color map index in the 64x64 array "pattern". */
     for (i=0; i<pattern_size; i++){
      for (j=0; j<pattern_size/16; j++){
       fscanf(fp,"%hx",&bitword);
        for (m=0; m<16; m++){
         if ((bitword&0x8000)>0){
          pattern[i][j*16+m]=next_color;
         }
         bitword=bitword<<1;
        }
      }
     }

     /* Advance to next index in color map */
     next_color+=1;
    }
    fclose(fp);
   }
   return(number_of_colors);
}
```

```
/*****************************************************************************
 ROUTINES:      polcolorf3i, polcolorf3s, polcolorf3
*****************************************************************************
 The functions polcolorf3i, polcolorf3s, and polcolorf3 will fill
 an arbitrary convex polygon with the texture pattern stored in
 "pattern". The argument "rotate", is the angle the polygon will
 be rotated about the z axis during mapping of the polygon to the
 x-y plane. The arguments of "horz" and "vert" position the reference
 point of the texture pattern. The argument "incr" specifies the polygon
 size for each texture point.
*****************************************************************************/

polcolorf3i(pattern_size,rotate,horz,vert,incr,pattern,n,parray)
short pattern_size;             /* 16 (16x16),32 (32x32),64 (64x64) */
Angle rotate;                   /* Angle of rotation about z axis */
int horz;                       /* Horizontal positioning of reference point*/
int vert;                       /* Vertical positioning of reference point */
Iccord incr;                    /* Size of texture point polygon */
Colorindex pattern[64][64];     /* Array of color indices */
long n;                         /* Number of points in polygon */
Icoord parray[][3];             /* Array of polygon points */
{
 register Coord xincr,yincr;
 Coord xmin,ymin,xmax,ymax;
 Coord fparray[30][3];
 register int i,j;
 int i_begin,j_begin;
 Matrix trans;

 /* Save attributes before redefining linestye, linewdth,etc. */
 pushattributes();
 set_up();

 /* Convert long integer array points to floating point. */
 for (i=0; i<n; i++){
   fparray[i][0]=(Coord)(parray[i][0]);
   fparray[i][1]=(Coord)(parray[i][1]);
   fparray[i][2]=(Coord)(parray[i][2]);
 }

 /* Map polygon points to x-y plane and rotate about z axis.
    Return the transformation matrix "trans" to map each texture
    point polygon to object space. */
 map_to_xy_plane(trans,rotate,n,fparray);

 /* Determine xmin, ymin, xmax and ymax of the polygon after
    it has been placed in the x-y plane to define upper
    and lower limits of texture pattern. */
 texture_pattern_limits(n,fparray,&xmin,&ymin,&xmax,&ymax);
```

```
/* Using the argument "incr", clipping lines parallel to x and y
   axis are calculated for each texture point to be used by
   clip_and_fill to fill texture point polygons. The next texture
   pattern color index is also calculated and passed on to
   clip_and_fill. */

i_begin=(pattern_size-vert)%pattern_size;
j_begin=(pattern_size-horz)%pattern_size;
i=i_begin;  j=j_begin;
yincr=0.0;

while ((ymin+yincr)<=ymax){
 xincr=0.0;
 while ((xmin+xincr)<=xmax){

  /* Calculate texture point polygon coordinates, apply transformation
     matrix "trans", and then fill the polygon. A value of 0 in the array
     "pattern" indicates transparent.*/
  if (pattern[i][j]!=0)
   clip_and_fill(trans,pattern[i][j],xmin+xincr,ymin+yincr,
          xmin+xincr+incr,ymin+yincr+incr,n,fparray);
  j+=1;
  xincr+=incr;

  /* If last column of texture pattern reached, repeat */
  if (j==pattern_size) j=0;
 }
 j=j_begin;
 i+=1;

 /* If last row of texture pattern reached, repeat */
 if (i==pattern_size) i=0;
 yincr+=incr;
}

/* Restore attributes */
popattributes();
}
```

```
/*****************************************************************************/

polcolorf3s(pattern_size,rotate,horz,vert,incr,pattern,n,parray)
short pattern_size;          /* 16 (16x16),32 (32x32),64 (64x64) */
Angle rotate;                /* Angle of rotation about z axis */
int horz;                    /* Horizontal positining of reference point */
int vert;                    /* Vertical positioning of reference point */
Scoord incr;                 /* Size of texture point polygon */
Colorindex pattern[64][64];  /* Array of color indices */
long n;                      /* Number of points in polygon */
Scoord parray[][3];          /* Array of polygon points */
{
 register Coord xincr,yincr;
 Coord xmin,ymin,xmax,ymax;
 Coord fparray[30][3];
 register int i,j;
 int i_begin,j_begin;
 Matrix trans;

 /* Save attributes before redefining linestyle, linewidth,etc. */
 pushattributes();
 set_up();

 /* Convert short integer array points to floating point */
 for (i=0; i<n; i++){
  fparray[i][0]=(Coord)(parray[i][0]);
  fparray[i][1]=(Coord)(parray[i][1]);
  fparray[i][2]=(Coord)(parray[i][2]);
 }

 /* Map polygon points to x-y plane and rotate about z axis.
    Return the transformation matrix "trans" to map each texture
    point polygon to object space. */
 map_to_xy_plane(trans,rotate,n,fparray);

 /* Determine xmin, ymin, xmax and ymax of the polygon to define upper
    and lower limits of the texture pattern. */
 texture_pattern_limits(n,fparray,&xmin,&ymin,&xmax,&ymax);

 /* Using the argument "incr", clipping lines parallel to x and y axis
    are calculated for each texture point to be used by clip_and_fill
    to fill texture point polygons. The next texture pattern color
    index is also calculated and passed on to clip_and_fill. */

 i_begin=(pattern_size-vert)%pattern_size;
 j_begin=(pattern_size-horz)%pattern_size;
 i=i_begin;   j=j_begin;
 yincr=0.0;
```

```
while ((ymin+yincr)<=ymax){
 xincr=0.0;
 while ((xmin+xincr)<=xmax){

  /* Calculate texture point polygon coordinates, apply transformation
    matrix "trans", and then fill the polygon. A value of 0 in the array
    "pattern" indicates transparent. */
  if (pattern[i][j]!=0)
   clip_and_fill(trans,pattern[i][j],xmin+xincr,ymin+yincr,
          xmin+xincr+incr,ymin+yincr+incr,n,fparray);
  j+=1;
  xincr+=incr;

  /* If last column of texture pattern reached, repeat */
  if (j==pattern_size) j=0;
 }
 j=j_begin;
 i+=1;

 /* If last row of texture pattern reached, repeat */
 if (i==pattern_size) i=0;
 yincr+=incr;
}

/* Restore attributes */
popattributes();
}
```

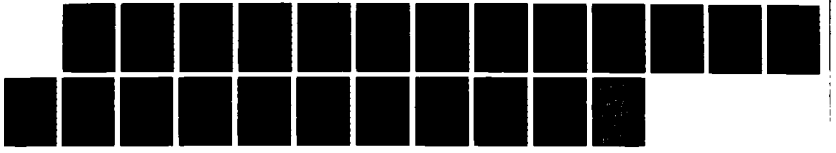
```
/*****************************************************************************/

polcolorf3(pattern_size,rotate,horz,vert,incr,pattern,n,parray)
short pattern_size;          /* 16 (16x16),32 (32x32),64 (64x64) */
Angle rotate;                /* Angle of rotation about z axis */
int horz;                    /* Horizontal positioning of referennce point */
int vert;                    /* Vertical positioning of reference point */
Coord incr;                  /* Size of texture point polygon */
Colorindex pattern[64][64];  /* Array of color indices*/
long n;                      /* Number of points in polygon */
Coord parray[][3];           /* Array of polygon points */
{
 register Coord xincr,yincr;
 Coord xmin,ymin,xmax,ymax;
 registr int i,j;
 int i_begin,j_begin;
 Matrix trans;

 /* Save attributes before redefining linestyle,linewidth,etc. */
 pushattributes();
 set_up();

 /* Map polygon points to x-y plane and rotate about z axis.
   Return the transformation matrix "trans" to map each texture
   point polygon to object space. */
 map_to_xy_plane(trans,rotate,n,parray);

 /* Determine xmin, ymin, xmax and ymax of polygon to define upper
   and lower limits of texture pattern. */
 texture_pattern_limits(n,parray,&xmin,&ymin,&xmax,&ymax);

 /* Using the argument "incr", clipping lines parallel to x and y
   axis are calculated for each texture point to be used by
   clip_and_fill to fill texture point polygons. The next texture
   pattern color index is also calculated and passed on to
   clip_and_fill. */

 i_begin=(pattern_size-vert)%pattern_size;
 j_begin=(pattern_size-horz)%pattern_size;
 i=i_begin;   j=j_begin;
 yincr=0.0;

 while ((ymin+yincr)<=ymax){
  xincr=0.0;
  while ((xmin+xincr)<=xmax){

   /* Calculate texture point polygon coordinates, apply transformation
     matrix "trans", and then fill the polygon. A value of 0 in the array
     "pattern" indicates transparent. */
   if (pattern[i][j]!=0)
    clip_and_fill(trans,pattern[i][j],xmin+xincr,ymin+yincr,
            xmin+xincr+incr,ymin+yincr+incr,n,parray);
```

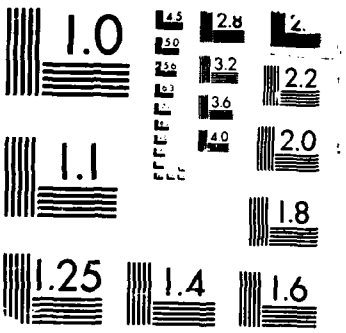
```
    j+=1;
    xincr+=incr;

    /* If last column of texture pattern reached, repeat */
    if (j==pattern_size) j=0;
   }
   j=j_begin;
   i+=1;

   /* If last row of texture pattern reached, repeat */
   if (i==pattern_size) i=0;
   yincr+=incr;
  }

 /* Restore attributes */
 popattributes();
}
```

```
/*****************************************************************************
 ROUTINE:      set_up
 *****************************************************************************
 The function set_up will set the system up for drawing the polygons
 representing each point in the texture pattern.
 *****************************************************************************/

set_up()
{
 linewidth(1);
 setlinestyle(0);
 lsrepeat(1);
 lsbackup(0);
}

/*****************************************************************************
 ROUTINE:      texture_pattern_limits
 *****************************************************************************
 The function texture_pattern_limits determines the upper and lower limits
 of a polygon in the xy plane. The lower limit is the smallest x and y values
 of all polygon coordinates and the upper limit is the largest x and y values
 of all polygon coordinates.
 *****************************************************************************/

texture_pattern_limits(n,parray,xmin,ymin,xmax,ymax)
long n;
Coord parray[][3];
Coord *xmin,*ymin,*xmax,*ymax;
{
 long i;
 *xmax=parray[0][0]; *ymax=parray[0][1];
 *xmin=(*xmax);      *ymin=(*ymax);
 for (i=1; i<n; i++){
  if (parray[i][0]>(*xmax))
   *xmax=parray[i][0];
  if (parray[i][1]>(*ymax))
   *ymax=parray[i][1];
  if ((*ymin)>parray[i][1])
   *ymin=parray[i][1];
  if ((*xmin)>parray[i][0])
   *xmin=parray[i][0];
 }
}
```

AD-A194 353 INVESTIGATION INTO THE USE OF TEXTURING FOR REAL-TIME COMPUTER ANIMATION(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA T W MEIER ET AL. MAR 88 NPS-52-88-003 2/2

UNCLASSIFIED F/G 12/6 NL





```
/*****************************************************************************
 ROUTINE:      clip_and_fill
*****************************************************************************
 The function clip_and_fill,used in polcolorf3,polcolorf3s,and polcolorf3i,
 creates and then fills the polygons representing each point in the texture
 pattern. Using the coordinates of the overall polygon and clipping
 boundaries of "xmin","ymin","xmax",and "ymax", the coordinates of a texture
 point polygon are calculated and then the polygon filled using the color
 with color map index "index". The transformation matrix "t" is passed in to
 map texture point polygons to their proper position in object space.
*****************************************************************************/

clip_and_fill(t,index,xmin,ymin,xmax,ymax,n,parray)
Matrix t;       /* Reverse transformation matrix for texture polygons*/
Colorindex index;  /* Color of next texture point polygon */
Coord xmin,ymin,xmax,ymax;    /* Clipping boundaries */
long n;         /* Number of points in overall polygon */
Coord parray[][3]; /* Array of overall polygon points */
{
 Coord boundaries[4];
 link head=NIL,p,q,r;
 long i,j;
 Coord yint,xint;
 long number_of_points;
 link intr,firstint,lastint;
 short firstdir,dir;
 Coord x,y,z;
 vertices points[50];
 int count=0;

 /* Boundaries used to clip polygon. */
 boundaries[0]=xmin;   boundaries[1]=ymin;
 boundaries[2]=xmax;   boundaries[3]=ymax;

 /* Put polygon points into circular linked list. */
 head=(&points[0]);
 head->xvertex=parray[0][0];
 head->yvertex=parray[0][1];
 head->next=NIL;
 p=head;
 for (i=1; i<n; i++){
  count+=1;
  p->next=(&points[count]);
  p=p->next;
  p->xvertex=parray[i][0];
  p->yvertex=parray[i][1];
  p->next=NIL;
 }
 p->next=head;

 /* Clip the polygon. */
 number_of_points=n;
```

```
/* Repeat for each clipping boundary */
for (j=0; j<4; j++){
 firstint=NIL;  lastint=NIL;
 n=number_of_points;
 q=head;

 /* Repeat for number of points in polygon */
 for (i=0; i<n; i++){
  p=q;
  if (number_of_points>1)
   q=p->next;

  /* Reduce number_of_points by one if both points p and
    q are not visible. */
  if (not_visible(j,boundaries[j],p,q))
   number_of_points-=1;

  /* Find intersection if one of the points is visible. */
  else if (partially_visible(j,boundaries[j],p,q)){
   find_intersection(j,boundaries[j],p,q,&xint,&yint);
   count+=1;
   r=(&points[count]);
   r->xvertex=xint;
   r->yvertex=yint;

   /* p is the point that is visible */
   if (p_inside(j,boundaries[j],p)){
    dir=0;

    /* p is the intersection */
          if (r->yvertex==p->yvertex&&r->xvertex==p->xvertex)
           intr=p;

    /* Insert intersection into linked list, increment
      number_of_points. */
          else{
           intr=r;
           p->next=r;
           number_of_points+=1;
    }
   }

   /* q is the point that is visible */
   else{
          dir=1;

    /* q is the intersection, decrement number_of_points */
          if (r->yvertex==q->yvertex&&r->xvertex==q->xvertex){
           intr=q;
           number_of_points-=1;
    }
```

```
        /* Insert intersection into linked list */
                else{
                 intr=r;
                 r->next=q;
        }
       }
       /* Save direction of polygon with first intersection */
       if (firstint==NIL){
        firstint=intr;  firstdir=dir;
       }
       else
        lastint=intr;
     }
    }
    /* Determine head pointer to new polygon. */
    if (firstint!=NIL&&lastint!=NIL){
     if (firstdir==0){
                firstint->next=lastint;
                head=firstint;
     }
     else{
                lastint->next=firstint;
                head=lastint;
     }
    }
    else if (firstint!=NIL&&lastint==NIL)
     head=firstint;
  }
  p=head;

  /* Draw the new filled polygon that represents a texture point. */
  if (number_of_points>0){
   color(index);
   x=p->xvertex; y=p->yvertex; z=parray[0][2];
   p->xvertex=x*t[0][0]+y*t[1][0]+parray[0][2]*t[2][0]+t[3][0];
   p->yvertex=x*t[0][1]+y*t[1][1]+parray[0][2]*t[2][1]+t[3][1];
   p->zvertex=x*t[0][2]+y*t[1][2]+parray[0][2]*t[2][2]+t[3][2];
   pmv(p->xvertex,p->yvertex,p->zvertex);
   p=p->next;
   for (i=1; i<number_of_points; i++){
    x=p->xvertex; y=p->yvertex; z=parray[0][2];
    p->xvertex=x*t[0][0]+y*t[1][0]+parray[0][2]*t[2][0]+t[3][0];
    p->yvertex=x*t[0][1]+y*t[1][1]+parray[0][2]*t[2][1]+t[3][1];
    p->zvertex=x*t[0][2]+y*t[1][2]+parray[0][2]*t[2][2]+t[3][2];
    pdr(p->xvertex,p->yvertex,p->zvertex);
    p=p->next;
   }
   pclos();
  }
}
```

```
/*****************************************************************************
 ROUTINE:    not_visible
*****************************************************************************
 The function not_visible, used in the function clip_and_fill
 determines the visibility of a line segment after application
 of a clipping boundary.
*****************************************************************************/

not_visible(boundary,limit,p_pt,q_pt)
long boundary;
Coord limit;
link p_pt;
link q_pt;
{
 switch(boundary)
 {
  case 0:
            return(p_pt->xvertex<limit&&q_pt->xvertex<limit);
            break;
  case 1:
            return(p_pt->yvertex<limit&&q_pt->yvertex<limit);
            break;
  case 2:
            return(p_pt->xvertex>limit&&q_pt->xvertex>limit);
            break;
  case 3:
            return(p_pt->yvertex>limit&&q_pt->yvertex>limit);
            break;
 }
}
```

```
/*****************************************************************************
 ROUTINE:      partially_visible
 *****************************************************************************
 The function partially_visible, used in clip_and_fill,
 determines the partial visibility of a line segment after
 application of a clipping boundary.
 *****************************************************************************/

partially_visible(boundary,limit,p_pt,q_pt)
long boundary;
Coord limit;
link p_pt;
link q_pt;
{
 switch(boundary)
 {
  case 0:
            return((p_pt->xvertex>=limit&&q_pt->xvertex>=limit)==0);
            break;
  case 1:
            return((p_pt->yvertex>=limit&&q_pt->yvertex>=limit)==0);
            break;
  case 2:
            return((p_pt->xvertex<=limit&&q_pt->xvertex<=limit)==0);
            break;
  case 3:
            return((p_pt->yvertex<=limit&&q_pt->yvertex<=limit)==0);
            break;
 }
}
```

```
/*****************************************************************************
 ROUTINE:      find_intersection
 *****************************************************************************
 The function find_intersection will find the intersection of
 a line segment and a clipping boundary.
 *****************************************************************************/

find_intersection(boundary,limit,p_ptr,q_ptr,xint,yint)
long boundary;
Coord limit;
link p_ptr;
link q_ptr;
Coord *xint;
Coord *yint;
{
 if (boundary==0||boundary==2){
   *xint=limit;
   *yint=((p_ptr->yvertex-q_ptr->yvertex)/(p_ptr->xvertex-q_ptr->xvertex))*
    limit+(p_ptr->xvertex*q_ptr->yvertex-p_ptr->yvertex*q_ptr->xvertex)/
    (p_ptr->xvertex-q_ptr->xvertex);
 }
 else{
   *yint=limit;
   *xint=((p_ptr->xvertex-q_ptr->xvertex)/(p_ptr->yvertex-q_ptr->yvertex))*
    limit+(p_ptr->yvertex*q_ptr->xvertex-p_ptr->xvertex*q_ptr->yvertex)/
    (p_ptr->yvertex-q_ptr->yvertex);
 }
}
```

```
/*****************************************************************************
 ROUTINE:      p_inside
 *****************************************************************************
 The function p_inside, used in the function clip_and_fill, determines
 if the first point in the line segment of a polygon is visible after
 application of a clipping boundary.
 *****************************************************************************/

p_inside(boundary,limit,p_pt)
long boundary;
Coord limit;
link p_pt;
{
 switch(boundary)
 {
  case 0:
            return(p_pt->xvertex>=limit);
            break;
  case 1:
            return(p_pt->yvertex>=limit);
            break;
  case 2:
            return(p_pt->xvertex<=limit);
            break;
  case 3:
            return(p_pt->yvertex<=limit);
            break;
 }
}
```

```
/*****************************************************************************
 ROUTINE:    map_to_xy_plane
 *****************************************************************************
 The function map_to_xy_plane takes the coordinates of the polygon to be filled
 with the texture pattern, and maps them to the x-y plane to facilitate
 placement of the texture pattern. At the same time, an inverse transformation
 matrix,"map_matrix", is built so that each texture point polygon can be
 mapped back to its proper position.
 *****************************************************************************/

map_to_xy_plane(map_matrix,rot,total_pts,pts)
Matrix map_matrix;  /* Inverse transformation matrix */
Angle rot;          /* Rotation angle about z axis */
long total_pts;     /* Total points in polygon */
Coord pts[][3];     /* Polygon coordinates */
{
 int pt1,pt2,pt3;
 float A,B,C,D;
 float anglex,angley,anglez;
 Matrix transform1,transform2,transform3,transform4;
 Matrix new_matrix;

 /* Get 3 non_colinear vertices from the polygon to be used for
    calculating the normal to the polygon. */
 get_3_pts(total_pts,pts,&pt1,&pt2,&pt3);

 /* Calculate the coefficients for the equations to the plane
    of the polygon and the normal to that plane. */
 calc_coeffs(total_pts,pts,pt1,pt2,pt3,&A,&B,&C,&D);

 /* Check to ensure all vertices of the polygon lie in the same
    plane. If so, then calculate the necessary angles of rotation. */
 if (planar_polygon(total_pts,pts,A,B,C,D)){

  /* Calculate the rotation about the y axis necessary to bring
     the normal into the y-z plane. */
  angles(A,B,C,'y',&anglex,&angley);

  /* Create the transformation matrix necessary to multiply with
     the array of vertices to generate the rotation as well as its
     inverse. */
  build_trans_matrix(transform1,transform2,angley,'y',pts,pt1);

  /* Perform the matrix multiplication with the array of
     vertices to form the new vertices. */
  map_points(total_pts,pts,transform1);

  /* Now extract three non_colinear vertices from the new array
     of vertices. */
  get_3_pts(total_pts,pts,&pt1,&pt2,&pt3);
```

```
/* Calculate the coefficients for the equations to the plane
   of the polygon and normal to that plane again. */
calc_coeffs(total_pts,pts,pt1,pt2,pt3,&A,&B,&C,&D);

/* Calculate the rotation about the x axis. */
angles(A,B,C,'x',&anglex,&angley);

/* Create the transformation matrix as well as its inverse. */
build_trans_matrix(transform1,transform3,anglex,'x',pts,pt1);

/* Perform the matrix multiplication. */
map_points(total_pts,pts,transform1);

/* Calculate the rotation about the z axis. */
rotate_about_z(rot,pts,&anglez);

/* Create the transformation matrix as well as its inverse. */
build_trans_matrix(transform1,transform4,anglez,'z',pts,pt1);

/* Perform the matrix multiplication.*/
map_points(total_pts,pts,transform1);

/* Using the inverse transformations calculated, create the transformation
   matrix to map each texture point polygon to object space. */
mult_matrix(new_matrix,transform3,transform2);
mult_matrix(map_matrix,transform4,new_matrix);
} }
```

```
/*****************************************************************************
 ROUTINE:    get_3_pts
 *****************************************************************************
 The function get_3_pts searches the array of vertices to find
 three points which do not lie on the same line and returns the indices
 of those points.
 *****************************************************************************/

get_3_pts(total_pts,pts,pt1,pt2,pt3)
long total_pts;
Coord pts[][3];
int *pt1,*pt2,*pt3;
{
 long count;
 int finished;
 *pt1=0; *pt2=0; *pt3=0;
 count=0; finished=0;

 /* The first vertex in the array (index=0) is selected for the first
    point. */
 while (count<total_pts&&!finished){

  /* Search the rest of the array of vertices to locate two more. */
  count+=1;
  if (*pt2==0){

   /* If the second point has not been selected yet, then we check
      to see if the coordinates of the vertex currently under exam
      are different. */
   if (pts[*pt1][0]!=pts[count][0]||
       pts[*pt1][1]!=pts[count][1]||
       pts[*pt1][2]!=pts[count][2])

     /* If so, then the vertex becomes the second point. */
     *pt2=count;
  }
  else if (*pt3==0){

   /* If the third point has not been selected yet, then we check
      to see if the coordinates of the vertex currently under exam
      are different from both the first and second point. */
   if ((pts[*pt1][0]!=pts[count][0]||
       pts[*pt1][1]!=pts[count][1]||
       pts[*pt1][2]!=pts[count][2])&&
      (pts[*pt2][0]!=pts[count][0]||
       pts[*pt2][1]!=pts[count][1]||
       pts[*pt2][2]!=pts[count][2]))

     /* If so, then the vertex becomes the third point. */
     *pt3=count;
```

```
    }
    else
      finished=1;
  }
  return;
}
```

```
/*****************************************************************************
 ROUTINE:    planar_polygon
 *****************************************************************************
 The function planar_polygon takes each vertex of the polygon and uses
 it to solve the equation for the plane as calculated earlier. If every
 vertex lies in the plane, then a value of 1 is returned.
 *****************************************************************************/

planar_polygon(total_pts,pts,A,B,C,D)
long total_pts;
Coord pts[][3];
float A,B,C,D;
{
 long count;
 int planar;
 count=0; planar=1;
 while (count<total_pts&&planar){

  /* The equation for the plane is setup to equal zero if the coordinates
     of the vertex lies in the plane. A range of .02 on either side of
     zero is allowed to handle round off error. This test is done for
     each vertex of the polygon. If any vertex fails the test then the
     function returns 0 for non_planar. */
  if ((A*(pts[count][0])+B*(pts[count][1])+C*(pts[count][2])-D)>.02||
      (A*(pts[count][0])+B*(pts[count][1])+C*(pts[count][2])-D)<-.02)
     planar=0;
  count+=1;
 }
 return(planar);
}
```

```
/*****************************************************************************
 ROUTINE:     calc_coeffs
 *****************************************************************************
 The function calc_coeffs uses the three vertices selected from the
 polygon to calculate the coefficients that will be used in both the equation
 of the plane the polygon lies in and the normal to that plane.
*****************************************************************************/

calc_coeffs(total_pts,pts,pt1,pt2,pt3,A,B,C,D)
long total_pts;
Coord pts[][3];
int pt1,pt2,pt3;
float *A,*B,*C,*D;
{
 float x0,x1,x2,y0,y1,y2,z0,z1,z2;
 x0=pts[pt1][0];
 y0=pts[pt1][1];
 z0=pts[pt1][2];

 x1=pts[pt2][0];
 y1=pts[pt2][1];
 z1=pts[pt2][2];

 x2=pts[pt3][0];
 y2=pts[pt3][1];
 z2=pts[pt3][2];

 *A=(y0-y1)*(z2-z1)-(y2-y1)*(z0-z1);
 *B=(z0-z1)*(x2-x1)-(z2-z1)*(x0-x1);
 *C=(x0-x1)*(y2-y1)-(x2-x1)*(y0-y1);
 *D=(*A)*pts[pt1][0]+(*B)*pts[pt1][1]+(*C)*pts[pt1][2];
 return;
}
```

```
/*****************************************************************************
 ROUTINE:      angles
*****************************************************************************
 The function angles utilizes the coefficients determined from the normal
 to the polygon to determine the angles of rotation necessary about the x
 and y axes. These rotations are necessary to bring the polygon into the
 x-y plane to facilitate placement of the texture pattern. This routine is
 called the first time to calculate the rotation about the y axis. The
 second time it is called is to determine the x axis rotation. The parameter
 "axis" is used to pass which angle is to be calculated.
*****************************************************************************/

angles(A,B,C,axis,anglex,angley)
float A,B,C;
char axis;
float *anglex,*angley;
{
 float pi,degx,degy,degz;

 pi=3.1415926536;

 degx=acos(A/sqrt(A*A+B*B+C*C))*360/(2*pi);
 degy=acos(B/sqrt(A*A+B*B+C*C))*360/(2*pi);
 degz=acos(C/sqrt(A*A+B*B+C*C))*360/(2*pi);

 if (A==0&&B==0){

  /* Rotate polygon 180 degrees about y axis. */
  if(C>0)
    *angley=(-180.0);

  /* No rotation necessary. */
  else
    *angley=0.0;

  *anglex=0.0;
 }

 else if (A==0&&C==0){

  /* Rotate polygon 90 degrees about x axis. */
  if (B<0)
    *anglex=90.0;
  else
    *anglex=(-90.0);
  *angley=0.0;
 }

 else if (B==0&&C==0){
```

```
 /* Rotate polygon 90 degrees about y axis. */
 if (A<0)
  *angley=(-90.0);
 else
  *angley=90.0;
 *anglex=0.0;
}

else{

 /* Must calculate amount of rotation about x and y axis. */
 *anglex=acos(C/sqrt(B*B+C*C))*360/(2*pi);
 *angley=acos(C/sqrt(A*A+C*C))*360/(2*pi);
 if (axis=='y'){
  if (degx<90.0)
          *angley=180.0-(*angley);
  else
          *angley=(*angley)-180.0;
 }
 else{
  if (degy<90.0)
          *anglex=(*anglex)-180.0;
  else
          *anglex=180.0-(*anglex);
 }
}
return;
}
```

```
/*****************************************************************************
 ROUTINE:      rotate_about_z
*****************************************************************************
 The function rotate_about_z takes the first two vertices of the new
 polygon in the x-y plane,as well as the input to xyplane of "rot" to
 calculate the rotation about the z axis. This again is to facilitate the
 placement of the texture pattern by the user onto the polygon.
*****************************************************************************/

rotate_about_z(rot,pts,anglez)
Angle rot;
Coord pts[][3];
float *anglez;
{
 float pi,deg;
 pi=3.1415926536;

 deg=acos(pts[1][0]/sqrt(pts[1][0]*pts[1][0]+pts[1][1]*pts[1][1]))
    *360/(2*pi);

 if (pts[1][1]>0.0)
  *anglez=(-deg)+(rot/10);
 else if (pts[1][1]<=0.0)
  *anglez=deg+(rot/10);
 return;
}
```

```
/*****************************************************************************
 ROUTINE:     build_trans_matrix
 *****************************************************************************
 The function build_trans_matrix creates the matrices to map the polygon
 to the x-y plane as well as rotate it about the z axis. For each matrix
 created, its inverse is also passed out.
 *****************************************************************************/

build_trans_matrix(t1,t2,angle,axis,pts,pt1)
Matrix t1;
Matrix t2;
float angle;
char axis;
Coord pts[][3];
int pt1;
{
 float pi,deg;
 pi=3.1415926536;

 deg=angle*((2*pi)/360.0);

 if ((axis=='x')||(axis=='X')){

  /* Transformation matrix for rotation about x axis. */
  t1[0][0]=1.0; t1[0][1]=0.0;       t1[0][2]=0.0;      t1[0][3]=0.0;
  t1[1][0]=0.0; t1[1][1]=cos(deg);  t1[1][2]=sin(deg); t1[1][3]=0.0;
  t1[2][0]=0.0; t1[2][1]=(-sin(deg)); t1[2][2]=cos(deg); t1[2][3]=0.0;
  t1[3][0]=(-pts[pt1][0]);
  t1[3][1]=(-pts[pt1][1])*cos(deg)+pts[pt1][2]*sin(deg);
  t1[3][2]=(-pts[pt1][1])*sin(deg)-pts[pt1][2]*cos(deg);
  t1[3][3]=1.0;

  /* Its inverse.*/
  t2[0][0]=1.0; t2[0][1]=0.0;          t2[0][2]=0.0;       t2[0][3]=0.0;
  t2[1][0]=0.0; t2[1][1]=cos(deg);   t2[1][2]=(-sin(deg)); t2[1][3]=0.0;
  t2[2][0]=0.0; t2[2][1]=sin(deg); t2[2][2]=cos(deg); t2[2][3]=0.0;
  t2[3][0]=pts[pt1][0]; t2[3][1]=pts[pt1][1]; t2[3][2]=pts[pt1][2];
  t2[3][3]=1.0;
 }
 else if ((axis=='y')||(axis=='Y')){

  /* Transformation matrix for rotation about y axis. */
  t1[0][0]=cos(deg); t1[0][1]=0.0; t1[0][2]=(-sin(deg)); t1[0][3]=0.0;
  t1[1][0]=0.0;      t1[1][1]=1.0; t1[1][2]=0.0;         t1[1][3]=0.0;
  t1[2][0]=sin(deg); t1[2][1]=0.0; t1[2][2]=cos(deg);    t1[2][3]=0.0;
  t1[3][0]=(-pts[pt1][0])*cos(deg)-pts[pt1][2]*sin(deg);
  t1[3][1]=(-pts[pt1][1]);
  t1[3][2]=pts[pt1][0]*sin(deg)-pts[pt1][2]*cos(deg);
  t1[3][3]=1.0;
```

```
 /* Its inverse. */
 t2[0][0]=cos(deg);  t2[0][1]=0.0; t2[0][2]=sin(deg); t2[0][3]=0.0;
 t2[1][0]=0.0;       t2[1][1]=1.0; t2[1][2]=0.0;      t2[1][3]=0.0;
 t2[2][0]=(-sin(deg)); t2[2][1]=0.0; t2[2][2]=cos(deg);  t2[2][3]=0.0;
 t2[3][0]=pts[pt1][0]; t2[3][1]=pts[pt1][1]; t2[3][2]=pts[pt1][2];
 t2[3][3]=1.0;
}
else if ((axis=='z')||(axis=='Z')){

 /* Transformation matrix for rotation about z axis. */
 t1[0][0]=cos(deg);   t1[0][1]=sin(deg);  t1[0][2]=0.0;  t1[0][3]=0.0;
 t1[1][0]=(-sin(deg)); t1[1][1]=cos(deg);  t1[1][2]=0.0;  t1[1][3]=0.0;
 t1[2][0]=0.0;       t1[2][1]=0.0;      t1[2][2]=1.0;  t1[2][3]=0.0;
 t1[3][0]=(-pts[pt1][0])*cos(deg)+pts[pt1][1]*sin(deg);
 t1[3][1]=(-pts[pt1][0])*sin(deg)-pts[pt1][1]*cos(deg);
 t1[3][2]=(-pts[pt1][2]);
 t1[3][3]=1.0;

 /* Its inverse. */
 t2[0][0]=cos(deg);   t2[0][1]=(-sin(deg)); t2[0][2]=0.0;  t2[0][3]=0.0;
 t2[1][0]=sin(deg); t2[1][1]=cos(deg); t2[1][2]=0.0;  t2[1][3]=0.0;
 t2[2][0]=0.0;       t2[2][1]=0.0;      t2[2][2]=1.0;  t2[2][3]=0.0;
 t2[3][0]=pts[pt1][0]; t2[3][1]=pts[pt1][1]; t2[3][2]=pts[pt1][2];
 t2[3][3]=1.0;
 }
 return;
}
```

```
/*****************************************************************************
 ROUTINE:     map_points
 *****************************************************************************
 The function map_points will map the polygon vertices to their new
 coordinates using one of the transformation matrices created in
 build_trans_matrix.
 *****************************************************************************/

map_points(total_pts,pts,t1)
long total_pts;
Coord pts[][3];
Matrix t1;
{
 int i;
 Coord x,y,z;

 for (i=0; i<total_pts; i++){
  x=pts[i][0]; y=pts[i][1]; z=pts[i][2];
  pts[i][0]=x*t1[0][0]+y*t1[1][0]+z*t1[2][0]+t1[3][0];
  pts[i][1]=x*t1[0][1]+y*t1[1][1]+z*t1[2][1]+t1[3][1];
  pts[i][2]=x*t1[0][2]+y*t1[1][2]+z*t1[2][2]+t1[3][2];
 }
 return;
}
```

```
/*****************************************************************************
 ROUTINE:      mult_matrix
 *****************************************************************************
 The function mult_matrix will multiply two matrices together to form a
 new transformation matrix, used to create the inverse transformation
 necessary to map each texture point polygon to its proper position.
 *****************************************************************************/

mult_matrix(matrix1,matrix2,matrix3)
Matrix matrix1;
Matrix matrix2;
Matrix matrix3;
{
 int i,j;
 for (j=0; j<4; j++){
  for (i=0; i<4; i++){
   matrix1[i][j]=matrix2[i][0]*matrix3[0][j]+matrix2[i][1]*matrix3[1][j]+
          matrix2[i][2]*matrix3[2][j]+matrix2[i][3]*matrix3[3][j];
  }
 }
}
```

LIST OF REFERENCES

[1] Catmull, E., "A Subdivision Algorithm for Computer Display of Curved Surfaces," Doctoral Dissertation, University of Utah, Salt Lake City, 1974.

[2] Blinn, J. F. and Newell, M. E., "Texture and Reflection in Computer Generated Images," *Communications of the ACM Vol. 19, No. 10*, (October 1976).

[3] Crow, F., "Summed-Area Tables for Texture Mapping," *Computer Graphics (Proc. SIGGRAPH 77) Vol. 18, No. 3*, (July 1984).

[4] Schweitzer, D., "Artificial Texturing: An Aid to Surface Visualization," *Computer Graphics (Proc. SIGGRAPH 83) Vol. 17, No. 3*, (July 1983).

[5] Samek, M., Slean, C., and Weghorst, H., "Texture Mapping and Distortion in Digital Graphics," *The Visual Computer Vol. 2, No. 5*, (1986).

[6] Blinn, J. F., "Simulation of Wrinkled Surfaces," *Computer Graphics (Proc. SIGGRAPH 78) Vol. 12, No. 3*, (1978).

[7] Fournier, A., Fussell, D., and Carpenter, L., "Computer Rendering of Stochastic Models," *Communications of the ACM Vol. 25, No. 6*, (June 1982).

[8] Oka, M., Tsutsui, K., Ohba, A., Kurauchi, Y., and Tago, T., "Real-Time Manipulation of Texture Mapped Surfaces," *Computer Graphics (Proc. SIGGRAPH 87) Vol. 21, No. 4*, (July 1987).

[9] Pavlidis, Theo, in *Algorithms for Graphics and Image Processing*, (Computer Science Press, 1972).

[10] Amanatides, J., "Realism in Computer Graphics: A Survey," *CG&A Vol. 7, No. 1*, (1987).

Distribution List for Dr. Michael J. Zyda

Defense Technical Information Center, Cameron Station, Alexandria, VA 22314	2 copies
Library, Code 0142 Naval Postgraduate School, Monterey, CA 93943	2 copies
Center for Naval Analyses, 4401 Ford Avenue Alexandria, VA 22302-0268	1 copy
Director of Research Administration, Code 012, Naval Postgraduate School, Monterey, CA 93943	1 copy
Dr. Michael J. Zyda Naval Postgraduate School, Code 52, Dept. of Computer Science Monterey, California 93943-5100	200 copies
Mr. Bill West, HQ, USACDEC, Attention: ATEC-D, Fort Ord, California 93941	1 copy
John Maynard, Naval Ocean Systems Center, Code 402, San Diego, California 92152	1 copy
El Wells, Naval Ocean Systems Center, Code 443, San Diego, California 92152	1 copy
Roger Casey, Naval Ocean Systems Center, Code 84, San Diego, California 92152	1 copy

END

DATE

FILMED

8-88

DTIC